JAIN SLEE (JSLEE) 1.1 Specification, Final Release

Specification Leads:

Sun Microsystems, Inc.

David Ferry
OpenCloud

JAIN SLEE (JSLEE) 1.1 Specification, Final Release

Table of Contents

Chapter 1 Introduction

This is the specification of the JAIN™ Service Logic and Execution Environment (SLEE) architecture. This architecture defines a component model for structuring the application logic of communications applications as a collection of reusable object-oriented components, and for composing these components into more sophisticated services. The SLEE architecture also defines the contract between these components and the container that will host these components at runtime.

The SLEE specification supports the development of highly available and scaleable distributed SLEE specification compliant application servers, but does not mandate any particular implementation strategy. More importantly, applications may be written once, and then deployed on any application server that complies with the SLEE specification.

In addition to the application component model, the SLEE specification also defines the management interfaces used to administer the application server and the application components executing within the application server. It also defines a set of standard Facilities, like the Timer Facility, Trace Facility, and Alarm Facility. The SLEE specification also recommends design patterns in the appendix, these design patterns are optional.

1.1 Target audience

The target audiences for this specification are communication application developers, vendors of the communication application servers, and telecommunications product vendors in the areas of Softswitches, SIP Proxy servers, Call Agents, and traditional class 4 or 5 switches who wish to deploy services on their product which are developed by service creators following the JAIN SLEE specification.

1.2 Typographic and terminology conventions

The SLEE specification uses the following typographic conventions:

- Logical Entity.
  Words with the first character capitalized describe a logical or conceptual entity defined by the SLEE specification, e.g. Activity, Activity Context, SBB, Service, Profile, and Profile Table.
- `InterfaceOrClass`.
  A single word in a `fixed width font` and made up of concatenated words, with the first character of each word capitalized, identifies a Java class or interface, e.g. the `TimerID` interface, the `CallBlockingSbb` abstract class and the `FooService` abstract class.
- `deployment-descriptor-element`.
  A single word in a `fixed width font` and made up of lowercase word(s) joined by a hyphen identifies a deployment descriptor element, e.g. `sbb`, `sbb-classes`, `sbb-abstract-class-name`.
- *Added in 1.1*
  A section or paragraph denoted with "*Added in 1.1"* indicates that the interface, method, or feature being described has been added in the 1.1 specification.
- *Changed in 1.1*
  A section or paragraph denoted with "*Changed in 1.1*" indicates that the interface, method or feature being described has changed in the 1.1 specification.
- *Clarified in 1.1*
  A section or paragraph denoted with "*Clarified in 1.1*" indicates that the interface, method or feature being described has been clarified in the 1.1 specification.
- *Deprecated in 1.1*
  A section or paragraph denoted with "*Deprecated in 1.1*" indicates that the interface, method or feature being described has been deprecated in the 1.1 specification.

The SLEE specification uses the following terminology conventions:

- "<*entity*> entity" refers to an instance of a logical or conceptual *entity*, e.g. an SBB entity.
- "<*entity*> object" refers to a Java object that represents an *entity* and provides the *entity* with a Java object that can be invoked to manipulate the entity, e.g. Activity object.
- "`<class>` object" refers to a Java object and this Java object is an instance of the specified *class*, e.g. a `CallBlockingSbb` object, a `FooService` object.
- "`<interface>` object" refers to a Java object and this Java object implements the specified *interface*, e.g. a `TimerID` object.
- References to a method signature in this specification typically refers to the combination of return type, method name and arguments.

1.3 Compatibility with 1.0 specification

Added in 1.1.

The 1.1 specification is designed such that a developer who knows 1.0 only needs to read 1.1 to access new features and general improvements.

JAIN SLEE 1.1 provides binary compatibility for JAIN SLEE 1.0 SLEE Components. SLEE Components written to 1.0 run on 1.1 with no modification or recompilation required. This compatibility is limited to the classes and interfaces defined by the specification.

JAIN SLEE 1.1 provides source compatibility for JAIN SLEE 1.0 Management clients. Management clients compiled using the 1.0 API will need recompilation against the 1.1 API. This compatibility is limited to the classes and interfaces defined by the specification.

In general where 1.1 improves a contract which was defined by 1.0, the 1.0 contract is deprecated in favour of the 1.1 contract. There are two approaches to deprecation; the first is where an existing method has been deprecated and replaced with new methods in the same interface; the second is where an entire interface or API has been deprecated and replaced with a new interface or API.

Each chapter in this specification provides a description of relevant modifications between 1.0 and 1.1.

New versions of deployment descriptors are defined in JAIN SLEE 1.1 and these have a new DTD. The DTDs for 1.0 remain.

The following are the backwards compatibility requirements for a 1.1 specification compliant JAIN SLEE.

- A 1.1 specification compliant JAIN SLEE must also implement the SBB, Event, Profile Specification, and Service contracts specified in the 1.0 specification. A 1.1 compliant JAIN SLEE must use the deployment descriptor DTD referenced by the component's DTD to determine whether the component is a 1.0 component or a 1.1 component. For example, if a Profile Specification deployment descriptor references the 1.0 Profile Specification DTD, then the JAIN SLEE container must implement and maintain the 1.0 contract for this 1.0 Profile Specification.
- A 1.0 SBB must not reference or use a 1.1 Profile Specification. This must be enforced by a 1.1 JAIN SLEE.
- A 1.0 Service must not reference or use a 1.1 SBB. This must be enforced by a 1.1 JAIN SLEE.
- A 1.0 SBB must be able to reference and use a 1.1 Event.
- The 1.1 Resource Adaptor contract replaces the 1.0 contract. A 1.1 compliant JAIN SLEE must implement the 1.1 Resource Adaptor contract, and need not implement the 1.0 contract.

1.4 Acknowledgements

The JSLEE architecture is a broad effort that includes contributions from numerous people.

We would like to thank the expert group for their invaluable contributions, detailed feedback and efforts throughout the specification. The expert group is comprised of AePONA, Alcatel-Lucent, Charles Arm-

strong, Phelim O'Doherty of BEA Systems, Emil Ivov, Instituto de Telecomunicacoes, jNetX, Inc., Kapsch Carriercom, Mobilkom Austria AG, Net4Call A.S., NIST, Nokia Corporation, Nortel, NTT Data Corporation, Brian O'Neill, OpenCloud, Personeta, Inc., Bartosz Baranowski, Eduardo Martins, Ivelin Ivanov of Red Hat, Inc., SBC, Sun Microsystems, Inc., Telecom Italia, TrueTel Communications Inc., Vodafone Group Services Limited.

We would like to thank Steven Adams of OpenCloud for his efforts in API design, specification reviews and editing. We would like to thank the following individuals at OpenCloud for their input to the design of the specification and work on various aspects of the Reference Implementation - Steven Adams , Nick Earle, Ben Evans, Oliver Jowett, Perrin Morrow and David Page. We would also like to thank the TCK team of Phil Baker, Alex Jowett, Thomas Spall, Samuel Sun and Michael van der Gulik for the huge amount of work involved in producing a comprehensive TCK for such a large specification.

Thanks to Koryn Grant for his discussions on the formal model, and to Swee Lim for advice on various technical and non-technical issues.

1.5 Organization

This document is divided up into several chapters as follows:

- Chapter 2, 'Overview' discusses at a high level each of the concepts required in order to understand the JAIN SLEE architecture.
- Chapter 3, 'Putting it Together' discusses how a developer would implement a Service Building Block (SBB) component, compose SBB components into higher level SBB components and Service components, define components for custom event types and provisioned data, deploy and administer Service components.
- Chapter 4, 'Addresses' discusses addresses.
- Chapter 5, 'SBB Local Interface' discusses how an SBB component defines its local interface that is used by other SBBs to invoke this SBB in a synchronous manner.
- Chapter 6, 'The SBB Abstract Class' discusses the SBB abstract class and the life cycle of its instances.
- Chapter 7, 'Activity Objects, Activity Contexts, and Activity Context Interface Objects' discusses Activities and Activity objects, how Activities are represented in the SLEE by Activity Contexts, the Activity Context Interface objects that expose the attributes of an Activity Context and the relationships of these objects to other objects and entities.
- Chapter 8, 'Events' discusses the SLEE event architecture, defining events, and the semantics of firing and receiving events.
- Chapter 9, 'Transactions' discusses how the SLEE uses well-established transaction concepts to respond to unexpected failures during execution.
- Chapter 10, 'Profiles and Profile Specifications' discusses defining the schema and behavior of Profiles using Profile Specifications, the classes and interfaces that implement these Profile Specifications, and the life cycle of instances of these classes.
- Chapter 11, 'Usage Parameters' discusses defining counters and sample sets that are updated by SBB components and the operations available to manage usage parameter sets and the data contained within them.
- Chapter 12, 'Runtime Environment' discusses the application programming interfaces provided by the SLEE at runtime and the restrictions that the SLEE imposes on components running in the SLEE.
- Chapter 13, 'Facilities' discusses the standard Facilities defined by the SLEE specification.
- Chapter 14, 'Management' discusses the management interfaces exposed by the SLEE including those for subscription management, Service management, and management of the SLEE Facilities.

- Chapter 15, ‘Resource Adaptors’ discusses resources, resource types, and how resources interact with the SLEE.
- Chapter 16, ‘Libraries’ discusses libraries that provide common functionality to other components installed in the SLEE.
- Appendix A, ‘Event Router Formal Model’ discusses the desired behavior of the SLEE event router with a formal mathematical specification.
- Appendix B, ‘Event Router Pseudo Code’ discusses the desired behavior of the SLEE event router using pseudo code.
- Appendix C, ‘JCC Resource Adaptor Type’ discusses the recommended way to adapt and plug JCC resources into the SLEE.
- Appendix D, ‘JAIN SIP 1.2 Resource Adaptor Type’ discusses the recommended way to adapt and plug JAIN SIP resources into the SLEE.
- Appendix E, ‘JAIN TCAP Resource Adaptor Type’ discusses the recommended way to adapt and plug JAIN TCAP resources into the SLEE.
- Appendix F, ‘Integration with EJB applications’ discusses the recommended way to integrate JAIN SLEE with Enterprise JavaBeans

1.6 Goals

The Service Logic and Execution Environment (SLEE) architecture has the following goals:

- Define the standard component architecture for building distributed object-oriented communications applications in the Java™ programming language.
- Allow the development of distributable communication applications by combining components developed using tools from different vendors.
- Support simple application development. Application developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, and other complex low-level APIs.
- Adopt the Write Once, Run Anywhere™ philosophy of the Java programming language. Application components can be developed once, and then deployed on multiple platforms without recompilation or source code modification.
- Address the development, deployment, and runtime aspects of a communications application’s life cycle.
- Define a pluggable resource adaptor framework that allows resources to plugged into the SLEE environment in a portable fashion across different SLEE implementations.
- Define the interfaces that enable communication applications from multiple vendors to develop and deploy components that can interoperate at runtime.
- Compatible with the Java 2 Enterprise Edition platform. Both SLEE and J2EE servers should co-exist seamlessly in the network executing and collaborating on converged service functions.
- Compatible with the Java Management Extensions (JMX) specification. JMX can be used as an instrument that can be controlled by a JMX server.
- Compatible with other Java programming language APIs.

1.7 Scope

The 1.1 version of the SLEE specification defines the contract between the SLEE and Resource Adaptors thereby providing a standard API for the connection of external resources to the SLEE environment. An example resource can be any be protocol stack or other event based resource. This version also enhances

some of the commonly used functions such as profile update, transaction boundaries, connection to synchronous external resources and facilities and management capabilities.

The 1.1 version now requires J2SE 1.4 (or later compatible version), and JMX 1.2.1.

1.8 Relationship with Java 2 Platform Enterprise Edition

SLEE is a container developed for asynchronous event driven applications. J2EE is a platform containing multiple containers developed for building enterprise applications typical to data centric networks. SLEE is standardized to meet the needs of developers that build real-time event processing applications such as communication or other real time applications, for example AutoID or Mobile Gaming. SLEE is not developed to compete against or replace J2EE. The goal of SLEE is to replace proprietary SLEE's in the marketplace with a standardized application server for event processing. A primary focus of SLEE is to enable enterprise applications access to event driven networks in a standardized manner via standardized integration of the two environments. It is thought that integration of SLEE with J2EE will lead to converged services that blend for example, data and voice services.

1.9 Relationship with Open Mobile Alliance (OMA)

The Open Mobile Alliance is a standards organization whose mission is to develop specifications for basic building blocks needed for developing interoperable mobile services. These building blocks are referred to as service enablers, examples include location, messaging and charging. SLEE is a standardized application server that can be used to implement the OMA service enablers, as well as host services designed using OMA enablers. SLEE's architecture is aligned with the OMA Services Environment (OSE), which is OMA's services architecture. OMA enabler components' interfaces can be mapped to SLEE SBB's; the SLEE platform provides the mechanisms for co-ordination among SBB's and the required infrastructure stacks via resource adaptors.

1.10 Relationship with 3GPP Open Service Access (OSA)

3GPP is a standards organization that standardizes application interfaces for the telecommunications industry. A subset of the work focuses on OSA. Defined within the OSA effort is a variety of Service Control Functions (SCF's) that enable communication application development such as call control or user interaction. SLEE is a standardized application server that can be used to build these applications. The application interfaces defined for each of these SCF's can be plugged into the SLEE via the resource adapter architecture, application developers can then program applications using the standardized component model and framework defined by SLEE. OSA defines a client/gateway architecture model for third party access, however the SCF's can also be deployed inside the network as next generation Service Control Points (SCP's), i.e Intelligent Network (IN) evolution. SLEE can be used as a standardized application server to build all these network elements and is agnostic to the network architecture.

1.11 Relationship with SIP Servlet

SIP Servlet (JSR 116) is an extension to the HTTP Servlet programming model specific to SIP. SIP Servlet can be viewed as an extension to a SIP stack. SIP Servlet is specific to the SIP protocol. SLEE is a protocol agnostic application server. By plugging in a SIP resource adapter i.e. (JSR 32) to SLEE a SIP application server is created. Both SIP Servlet and SLEE with a SIP resource can be used to build out a SIP centric communications network. Advantages and disadvantages exist for each solution based on the network requirements, for example SLEE is component based and defines an explicit transaction model for concurrency control, SIP Servlet does not. The choice of SIP Servlet technology or SLEE technology is beyond the scope of this document, however both may co-exist in a SIP based network.

Chapter 2 Overview

A SLEE is an application server. This application server is a container for software components. The SLEE specification is designed and optimized for event driven applications.

The SLEE specification is targeted at users of a SLEE. These users include the developer of software components running on the SLEE and Administrators managing the software components deployed on the SLEE.

This specification does not specify how a SLEE vendor should implement a SLEE and the underlying technology. Multiple implementation choices allow multiple SLEE vendors to differentiate their products and to ensure diversity in the SLEE market.

2.1 Scope of the SLEE specification

The SLEE specification specifies:

- The SLEE component model and how it supports event driven applications as SLEE components.
- How SLEE components can be composed and invoke each other.
- How provisioned data is specified, externally managed and accessed by SLEE components.
- SLEE Facilities.
- How resources fit into the SLEE architecture and how SLEE applications interact with these resources.
- How events are routed to application components.
- The management interfaces of a SLEE.
- How applications are packaged for deployment into a SLEE.

2.1.1 Component model

The SLEE component model is targeted at event driven applications, also known as asynchronous applications. These applications receive requests in the form of events.

2.1.1.1 Events and event types

An event represents an occurrence that requires application processing. It carries information that describes the occurrence, such as the source of the event. An event may originate from:

- An external resource such as a communications protocol stack.
- Within the SLEE.
  The SLEE uses events to communicate changes in the SLEE that may be of interest to applications running in the SLEE. For example, the Timer Facility emits an event when a timer has expired. The SLEE also emits an event after an Administrator modifies an application's provisioned data.
- An application running in the SLEE.
  An applications running within the SLEE may use events to signal, invoke, or communicate with other applications running in the SLEE.

Every event in the SLEE has an event type. The event type of an event determines how the event is routed to different application components.

2.1.1.2 Event driven applications

An event driven application typically does not have an active thread of execution. Typically, an event driven application defines methods that are invoked when events are delivered to the application. These methods contain application code that inspect the event and perform additional processing to handle the

event. The application code may interact with the resource that emitted the event or other resources, fire new events or update the application state.

A common example of an event driven application is an application that implements a state machine. A state machine has states and transitions. A transition occurs when there is an occurrence that causes the state machine to move from one state to another state. A transition may be annotated with computation that occurs during the transition.

When an application implements a state machine, the application's variables determine the state of the state machine, Events represent occurrences on state transitions, and application code performs computations on transitions.

A common way to build an event driven application is to provide a single event handler method to receive all events. When the event handler method receives an event, it inspects the event and directs further processing of the event based on the event type. Often this processing is delegated to a separate method for each event type. The SLEE component model models the external interface of an event driven application as a set of events that the application can receive. Each event type is handled by its own event handler method. This enforces a well-defined event interface. The event driven component model allows the SLEE to provide the event routing logic for the application.

2.1.1.3 Application components

The SLEE architecture defines how an application can be composed of components. These components are known as Service Building Block (SBB) components.

Each SBB component identifies the event types accepted by the component and has event handler methods that contain application code that processes events of these event types. In addition, an SBB component may have an interface for synchronous method invocations.

At runtime, the SLEE creates instances of these components to process events and deletes components that are no longer eligible for event processing.

2.1.2 Provisioned data

The SLEE specification defines management interfaces and specifies how applications running in the SLEE access provisioned data. Typical provisioned data includes configuration data or per-subscriber data.

2.1.3 SLEE Facilities

The SLEE specification defines a number of Facilities that may be used by SBB, Profile Specification, and Resource Adaptor components. These Facilities include the Timer Facility, the Trace Facility, the Alarm Facility, the Event Lookup Facility, the Service Lookup Facility, the Profile Facility and the Activity Context Naming Facility. These facilities provide access to functionality whose implementation depends on the particular SLEE implementation in use. The behavior of the facilities is specified by this specification.

2.1.4 Resources

Changed in 1.1: Updated to introduce 1.1 resource adaptor concepts.

A resource represents a system that is external to a SLEE. Examples include network devices, protocol stacks, and databases. These resources may or may not have Java APIs. Resources with Java APIs define Java classes or interfaces to represent the events emitted by the resource. Example Java APIs include call agents supporting Java Call Control (JSR 21), Messaging services (JSR 212), various protocol stacks i.e. SIP (JSR 32) and databases supporting JDBC (JSR 54).

2.1.4.1 Resource Adaptors

The SLEE architecture defines how applications running within the SLEE interact with resources through resource adaptors. Resource adaptors adapt resources to the requirements of the SLEE. The SLEE architecture defines the following resource adaptor concepts.

- Resource adaptor type.
  A resource adaptor type declares the common characteristics for a set of resource adaptors. It defines the Java interfaces implemented by the resource adaptors of the same resource adaptor type. One of these interfaces is known as the resource adaptor interface. It also defines the event types fired by the resource adaptors of the same resource adaptor type. The SLEE specification includes non-normative recommendations for the JCC, SIP, and TCAP resource adaptor types. Typically, a resource adaptor type is defined by an organization of collaborating SLEE or resource vendors, such as the SLEE expert group.
- Resource adaptor.
  A resource adaptor is an implementation of a one or more resource adaptor types. There may be multiple implementations of the same resource adaptor type. A resource adaptor consists of a set of Java classes and a deployment descriptor. It must include a Java class that implements the resource adaptor interface of its resource adaptor type. Typically, a resource adaptor is provided either by a resource vendor or a SLEE vendor to adapt a particular resource implementation to a SLEE. It also includes the Java classes of the resource. For example, vendor A that has a SIP resource may provide a SIP resource adaptor (i.e. a resource adaptor of the SIP resource adaptor type) or SLEE vendor B may provide a SIP resource adaptor that adapts vendor A's SIP resource to its SLEE. In either case, the SIP resource adaptor will include the Java classes of the SIP implementation and the Java classes that adapt the SIP implementation to the SLEE.
- Resource adaptor entity.
  A resource adaptor entity is an instance of a resource adaptor. Multiple resource adaptor entities may be instantiated from the same resource adaptor, For example, a JCC resource adaptor that adapts a SIP-based JCC implementation to the SLEE may accept IP addresses and port numbers as parameters. The Administrator may instantiate one of these JCC resource adaptors for each unique IP address and port number pair. An important function of a resource adaptor entity is to forward events originating from the resource that the resource adaptor entity represents to the SLEE.
- Resource adaptor object.
  A resource adaptor object is an instance of a resource adaptor class. Resource adaptor objects are the Java objects that the SLEE uses to interact with resource adaptor entities.
- SLEE endpoint.
  A SLEE endpoint is the interface between the resource adaptor and the SLEE. It is implemented by the SLEE and is the entry point of resource adaptors to create new activities and fire events.

2.1.5 Event routing

The SLEE specification defines how an event emitted by an event producer is routed and delivered to one or more component instances interested in the event. A SLEE has a logical event router. This event router receives events emitted from all event producers and delivers events to multiple component instances interested in the event.

2.1.6 Management interfaces

The SLEE architecture defines management interfaces for managing the SLEE and components running in the SLEE. These management interfaces enable an Administrator to manage the SLEE, deploy and manage applications into the SLEE. It includes interfaces for adding, removing, and modifying provisioned data.

2.1.7 Packaging

An application is composed from one or more components. Each component may consist of one or more Java class files and deployment descriptors. The SLEE specification defines how a component's Java class files and the component's deployment descriptor are aggregated into a deployable unit.

2.1.8 Transactions

The SLEE utilizes transactions as its concurrency control mechanism. The SLEE transaction model extends the transaction model defined in the Java Transaction API with asynchronous capabilities. The SLEE specification only supports local transactions; however distributed transactions may be supported in a later release of this specification.

2.2 Main abstractions

The SLEE architecture defines the following core abstractions and concepts:

- Event and event type
- SBB component, SBB component graph, and root SBB component
- SBB entity, SBB entity tree, and root SBB entity
- Cascading removal of SBB entity sub-tree
- SBB abstract class and SBB object
- SBB local interface and SBB local object
- Activity, Activity object, and Activity Context
- How the SLEE reclaims an SBB entity tree and its associated descendent SBB entities
- Activity Context Interface interface and Activity Context Interface object
- Profile, Profile Table, and Profile Specification
- Service
- Resource Adaptor type and Resource Adaptor components
- Library component *(Added in 1.1)*
- Deployable unit
- Management interface

2.2.1 SBB component

An SBB component defines:

- Event types received and fired by the SBB component.
- Per-instance state.
  The per-instance state should be held in Container Managed Persistent (CMP) fields that can maintain persistent state that should persist across failures.
- Event methods.
  The SBB component provides an event handler method for each event type received by the SBB component. The event handler method contains application logic to process events of a specific event type. The SBB component also declares a fire event method for each event type fired by the SBB component.
- SBB local interface methods.
  The SBB component declares the SBB local interface of the SBB component. The SBB local interface specifies the methods of the SBB component that may be invoked synchronously. The SBB local interface methods of an SBB component instance can only be invoked by another SBB component instance within the same SBB component instance tree (see Section 2.2.4). The SBB also provides the implementation of the methods of the SBB local interface.
- Child relations.
  The SBB component may be related to zero or more child SBB components. The SBB component specifies its child SBB component relations. The SBB component identifies each child SBB component relation through a deployment descriptor element and declares a child relation accessor

method for runtime access to this relation (see Section 2.2.2). It also assigns a *default event delivery priority* to each child SBB component (see Section 2.2.5).

- Shareable data.
  The SBB component defines the data that it wishes to share with other components as a set of Activity Context attributes. Each Activity Context attribute has a name and a type. These attributes are stored in one or more Activity Contexts (see Chapter 7). An SBB component defines an SBB Activity Context Interface interface that provides type-safe accessor methods to get and set these attributes.

This document may also use *SBB* to refer to an SBB component.

2.2.2 SBB graph and root SBB

The SBB developer composes SBBs by specifying the *child relations* among SBBs. The parent and the child are the two roles in a relation. Logically, an SBB is a node in an SBB graph and a child relation is a directed edge from the node of the parent SBB to the node of the child SBB. Each edge has a label that indicates the default event delivery priority of the child relation.

For example, the SBB developer may develop three components, the X, Y, and Z SBB. The X SBB may have the Y SBB and the Z SBB as its children. The Y SBB may have the Z SBB as its child. The Z SBB may have Y SBB and itself as its own children. Figure 1 illustrates this SBB graph.

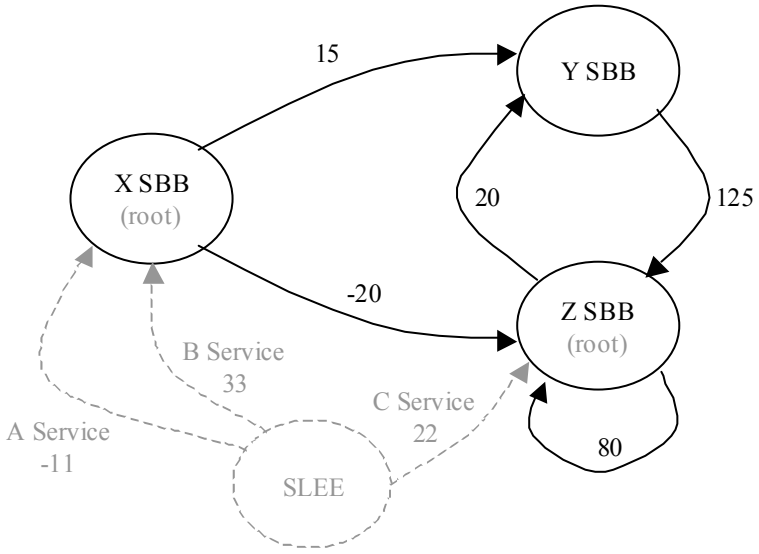

Figure 1 An SBB graph example

An SBB may be a parent to zero or more other SBBs including itself. In Figure 1, Z is a parent to Y and to itself. At the same time, it can also be a child to zero or more other SBB components including itself. Z is a child to X, Y and itself.

A root SBB is an SBB that may be instantiated by the SLEE to process events. A root SBB must declare a non-empty set of initial event types that may cause the SLEE to instantiate an instance of the SBB (see 8.6.1). These SBBs are known as root SBBs because their instances may become roots of SBB entity trees (see Section 2.2.4).

Typically, a root SBB represents a “complete service”. For example, the SBB developer may develop a CallBlocking SBB and a CallForwarding SBB to implement the “call blocking” service and the “call forwarding” service. These SBBs are root SBBs since instances of these SBBs may be instantiated to block and forward calls, respectively. The SBB developer may create a new root CallBlockingAndForwarding SBB to implement the “call blocking and forwarding” service from the CallBlocking SBB and CallForwarding SBB.

A parent SBB can be related to the same child SBB through multiple relations. Different priorities may be assigned to these relations. In the above Figure, the SLEE is the logical parent of the all root SBBs, i.e. there are two relations that relate the SLEE as the logical parent to the X SBB.

2.2.3 SBB entities

An *SBB entity* is an instance of an SBB component. An SBB entity is a logical entity that represents the persistent per-instance state of the instance.

2.2.4 SBB entity trees and root SBB entities

An SBB entity may create zero or more SBB entities at runtime. Since an SBB entity may only be created once, it has only a single parent.

An SBB entity tree is a directed acyclic graph that represents the child relations between SBB entities. A node in this tree represents an SBB entity. A directed edge represents the child relation from a parent SBB entity to a child SBB entity. The label on the directed edge indicates the event delivery priority of the child SBB entity relative to the child's siblings (see Section 2.2.5).

The SLEE will only create instances of root SBBs, known as root SBB entities. These SBB entities are root SBB entities because they are the root nodes of their respective SBB entity trees. An SBB entity belongs to exactly one SBB entity tree.

For example, the SLEE may instantiate a root CallBlockingAndForwarding SBB entity to handle a single call. This SBB entity may subsequently create zero or more child CallBlocking SBB or CallForwarding SBB entities to help it handle the call. This SBB entity tree has one root CallBlockingAndForwarding SBB entity and zero[1] or more directed edges from the root SBB entity to its child CallBlocking SBB entities or CallForwarding SBB entities.

To illustrate that a child SBB of a root SBB may also be a root SBB in this example, the CallBlocking SBB is also a root SBB. Hence, the SLEE may also create a root CallBlocking SBB entity to handle a different call. This SBB entity tree of this root CallBlocking SBB entity only contains the root CallBlocking SBB entity.

Figure 2 illustrates several SBB entity trees that may be instantiated from the SBB graph in Figure 1. It also shows the SLEE as the "logical parent" of all root SBB entities. X1, X2 and Z2 are the root SBB entities in this example. X1, Y1, Z1, and Y2 belong to the SBB entity tree rooted at X1. X2 belongs to the SBB entity tree rooted at X2. Z2, Z3, Y3, Z4, Z5, Z6 belong to the SBB entity tree rooted at Z2.

[1] It has zero directed edges before the root SBB entity creates any child SBB entities or after it has removed all its child SBB entities.

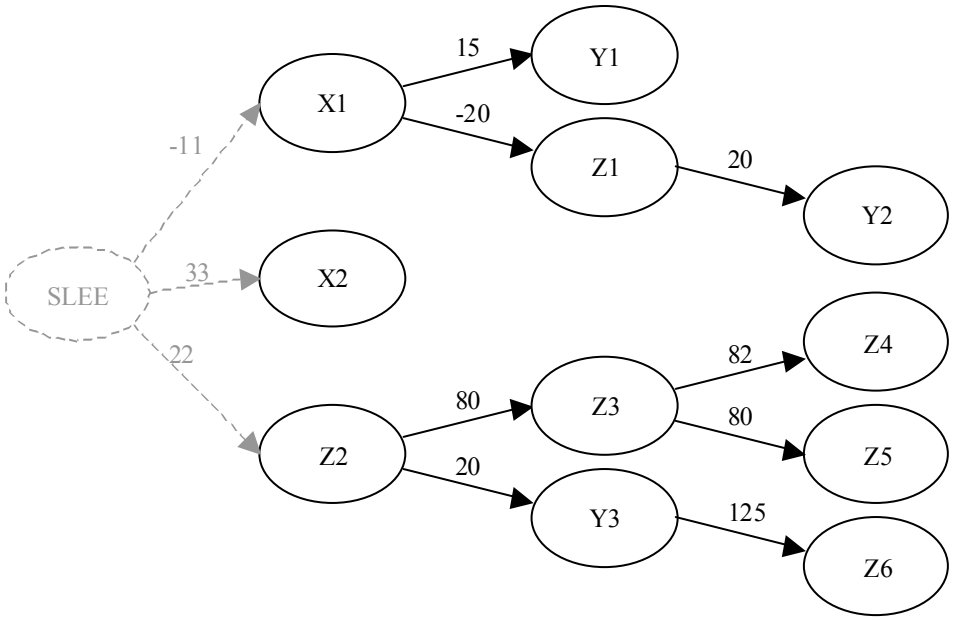

Figure 2 SBB entity trees examples

An SBB entity sub-tree is a tree whose root is a descendent of a root SBB entity. For example, Z3, Z4, and Z5 belong to the SBB entity sub-tree rooted at Z3.

2.2.5 Event delivery priority

An event is delivered in order to each SBB entity that is interested in receiving the event. A parent SBB entity always receives the same event before its child SBB entities. The event delivery priority of an SBB entity determines the order that sibling SBB entities receive the event.

The parent SBB specifies a default event delivery priority for each child relation. When a parent SBB entity creates a child SBB entity, the SLEE assigns the default event delivery priority specified by the child relation to the child SBB entity. At runtime, the event delivery priorities of SBB entities may be changed. In the SBB trees illustrated by Figure 2, all child SBB entities retain the default event delivery priority specified by its child relation, except SBB entity Z4's priority has changed to 82.

2.2.6 Cascading removal of SBB entity sub-tree

An SBB entity can be removed by invoking the remove method on the `SbbLocalObject` interface or via the `ChildRelationObject` interface. When this occurs, the SLEE performs the necessary operations to remove the SBB entity and all the SBB entity's descendents, i.e. the SLEE removes the sub-tree rooted at the SBB entity targeted for removal. This mechanism exists to ensure that SBB entities are not 'leaked' due to lost references to a particular SBB entity in SBB code[2].

When the SLEE removes a sub-tree, it performs a pre-order traversal of the sub-tree to delete the SBB entities in the sub-tree. It deletes the parent SBB entity before it deletes the child SBB entities. The SLEE does not specify the order in which sibling SBB entities are deleted. When the SLEE removes an SBB entity, the SLEE invokes the appropriate life cycle methods (see Section 6.3) on any SBB objects that cache the SBB entity's state and the SLEE detaches the SBB entity from all Activity Contexts (see Section 7.3.3) .

[2] JVM garbage collection does not solve this problem as SBB entity state is not directly related to instances of SBB objects.

In Figure 2, if X1 is removed, X1 is deleted before Y1 and Z1. Y1 may be deleted before Z1 or Z1 may be deleted before Y1. After Z1 is deleted, Y2 is deleted. The end result is that the sub-tree rooted at X1 is removed.

2.2.7 SBB object

An *SBB object* is an instance of the SLEE generated class that extends the SBB abstract class (see Section 3.1.4). The SBB abstract class is the SBB Developer provided Java class that contains the definitions and methods listed above.

The SLEE creates and manages a pool of SBB objects. At runtime, the SLEE may assign zero[3] or more SBB objects to represent an SBB entity. When an SBB object is assigned to represent an SBB entity, the SBB object is in the Ready state (see Section 6.3). It can receive and fire events, receive synchronous method invocations, and access and update the persistent state of the SBB entity. Another viewpoint is that the SBB object caches a copy of the persistent data of the SBB entity to provide transactional semantics (see Chapter 9).

2.2.8 SBB local interface and SBB local object

Each SBB has an SBB local interface. The SBB local interface of an SBB is either an SBB specific local interface provided by the SBB Developer that extends the `SbbLocalObject` interface, or the `SbbLocalObject` interface if the SBB Developer did not provide an SBB specific local interface.

An SBB object synchronously invokes the SBB object of a target SBB entity through an SBB local object. The SBB local object is a SLEE implemented object that implements the SBB local interface and represents the target SBB entity. When the SBB object invokes an SBB Developer defined method on the SBB local object, the corresponding method in the SBB abstract class that implements this method is invoked on an SBB object that represents the target SBB entity's state. Another viewpoint is that the SBB local object is a client side stub object for synchronous method invocations.

Logically, an SBB local object represents only one SBB entity, but zero or more SBB local objects may represent the same SBB entity.

An SBB object is only allowed to invoke SBB local objects that represent SBB entities in the same SBB entity tree. The execution behavior of the SLEE is undefined if an SBB object attempts to invoke an SBB local object that represents an SBB entity that is not in the same SBB entity tree.

2.2.9 Activity

An Activity represents a related stream of events. These events represent occurrences of significance that have occurred on the entity represented by the Activity. From a resource's perspective, an Activity represents an entity within the resource that emits events on state changes within the entity or resource.

For example, a phone call may be an Activity.

2.2.10 Activity object

An Activity object is a Java object that encapsulates an Activity and may provide methods that are used to interact with the Activity. Each resource adaptor type may define one or more types of Activity objects. In most cases, Activity objects are created and owned by resource adaptor entities.

For example, a `JccCall` object is an Activity object that represents a phone call (which is the Activity).

2.2.11 Activity Context

The SLEE uses an Activity Context to represent and encapsulate an underlying Activity object within the SLEE. An Activity Context is a logical entity within the SLEE. It does not have a visible Java API. There is a one-to-one relationship between an Activity object and an Activity Context.

---

[3] The SBB entity exists but the SLEE may not assign an SBB object to the SBB entity until the SBB entity is invoked.

- An Activity Context is also a store for attributes that may be shared by multiple SBB entities that interact with the Activity object represented by the Activity Context. These SBB entities can read and modify attributes stored in the Activity Context.
- An Activity Context is also an event channel that accepts events fired on the Activity Context and distributes these events to the SBB entities attached to the Activity Context. An SBB entity can invoke other SBB entities in an asynchronous manner by firing events on the Activity Context.

An SBB entity may be attached to one or more Activity Contexts. An SBB entity can only receive events fired on Activity Contexts that it is attached to.

The SLEE does not count the number of times an SBB entity is attached or detached from an Activity Context. The SBB entity is either attached to the Activity Context or not. Attaching multiple times consecutively is the same as attaching once and detaching multiple times is the same as detaching once.

The SLEE attaches and detaches SBB entities from Activity Contexts in the following situations.

- When a new root SBB entity is created to process its initial event, the SLEE attaches the new root SBB entity to the Activity Context on which the initial event was fired (see Section 8.6).
- After the underlying Activity object of an Activity Context ends, the SLEE detaches SBB entities attached to the Activity Context (see Section 7.3.3).
- Before the SLEE removes an SBB entity as part of a cascading removal of an SBB entity sub-tree, the SLEE detaches the SBB entity from all Activity Contexts.

An SBB entity may also explicitly invoke methods to change which Activity Contexts it or other SBB entities in the same SBB entity tree are attached to by explicitly attaching those SBB entities to these Activity Contexts or by explicitly detaching those SBB entities from these Activity Contexts. For example,

- An SBB entity may attach itself to the Activity Context of a new Activity object that it has created. This allows the SBB entity to receive events fired on the Activity Context by the newly created Activity object.
- An SBB entity may detach itself from an Activity Context if it no longer wants to receive events fired on that Activity Context.
- An SBB entity may explicitly attach another SBB entity to an Activity Context. This functionality allows the delegation of all or some processing of events fired on the Activity Context to another SBB entity.

2.2.12 How the SLEE reclaims an SBB entity tree

The SLEE uses the attachment count mechanism to remove SBB entity trees that are no longer attached to any Activity Contexts and hence will no longer be receiving events.

The SLEE maintains an implicit attachment count for each SBB entity. The attachment count of an SBB entity is the total number of SBB entity to Activity Context attachments that the SBB entity and its descendent SBB entities have.

Figure 3 illustrates attachment relations and counts graphically. In this graph, rectangles represent Activity Contexts and ovals represent SBB entities. An SBB entity to Activity Context attachment is represented by a non-directed edge between an SBB entity and an Activity Context. There is at most one edge between each SBB entity and Activity Context pair. The attachment count of an SBB entity is the total number of edges that connect the SBB entity and its descendents to Activity Contexts. The number in each oval is the attachment count of the SBB entity.

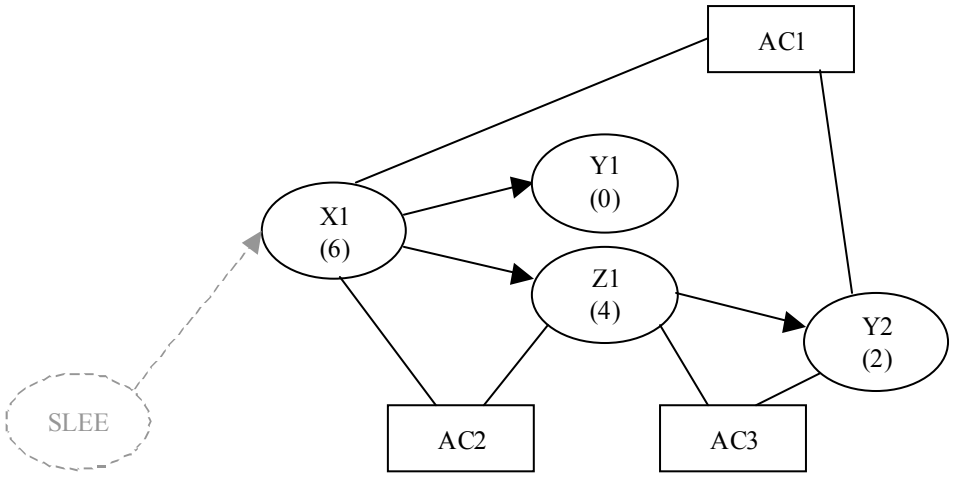

Figure 3 SBB attachments

The attachment count of a parent SBB entity is the sum of the attachment counts of all its child SBB entities and the number of Activity Contexts that the parent SBB entity is directly attached to.

When an Activity Context's Activity object ends, the SLEE detaches all SBB entities attached to the Activity Context (see Section 7.3.3). In a well-behaved system, Activity objects should end after a reasonable finite amount of time. Eventually, the attachment count of the root SBB entity will reach zero.

The SLEE uses the attachment count of the root SBB entity to find out when the root SBB entity and its descendents will no longer be receiving events (they must be attached to at least one Activity Context to receive events). If the SLEE finds the root SBB entity's attachment count to be zero after an invocation (such as delivering an event), the SLEE reclaims the SBB entity tree by removing the root SBB entity. Removing the root SBB entity initiates a cascading removal operation that also removes all the descendent SBB entities (see Section 2.2.6).

Detach operations are transactional (see Chapter 9 for a more detailed discussion on transactions). The effects of the detach operation are only visible within the transaction until the transaction commits. If the transaction does not commit, then the detach operation "did not" occur.

2.2.13 Activity Context reclamation and SBB entity tree removal example

In this example, there are two SBB entities in the SBB entity tree. There is a root SBB entity and the root SBB entity has a single child SBB entity. The root SBB entity is not attached to any Activity Context and the child SBB entity is attached to a single Activity Context.

The following simplified[4] example sequence illustrates how the SLEE and these SBB entities interact with the Activity Context to reclaim the Activity Context after its underlying Activity object ends (see Section 7.2.4 for details) and to remove the SBB entity tree after it is no longer attached to any Activity Context.

1. The child SBB entity invokes a method on the underlying Activity object of the Activity Context to end the Activity object (see Section 7.2.6). The SLEE moves the Activity Context to the Ending state.
2. The SLEE delivers an Activity End Event to the SBB entities attached to the Activity Context that can receive the Activity End Event, then detaches all SBB entities from the Activity Context. When the SLEE detaches the child SBB entity from the Activity Context, the SLEE decrements the attachment count of the child SBB entity from one to zero. It also decrements the attachment

[4] Not all steps are shown.

count of the ancestors of the child SBB. In this case, the SLEE decrements the attachment count of the root SBB entity from one to zero.

3. The SLEE moves the Activity Context to the Invalid state, reclaims it and then removes the SBB entity tree by initiating a cascading removal of the root SBB entity as the root SBB attachment count is zero.

2.2.14 Activity Context Interface interface and object

An SBB entity interacts with an Activity Context through an Activity Context Interface object. The Activity Context Interface object provides an Activity Context with a visible Java API that can be invoked. Through an Activity Context Interface object, an SBB can get access to the underlying Activity object of an Activity Context.

All Activity Context Interface objects implement the generic `ActivityContextInterface` interface and may optionally implement an extended SBB Activity Context Interface interface. The generic `ActivityContextInterface` interface does not declare any shareable attributes. There are two kinds of Activity Context Interface objects.

- Generic Activity Context Interface object.
  A generic Activity Context Interface object implements the generic `ActivityContextInterface` interface but does not implement an extended SBB Activity Context Interface interface. It cannot be used to access attributes stored in an Activity Context.
- SBB Activity Context Interface object.
  An SBB may define an SBB Activity Context Interface interface that extends the generic `ActivityContextInterface` interface. The SBB Activity Context Interface interface defines the SBB specific view of an Activity Context. An SBB Activity Context Interface object implements an SBB Activity Context Interface interface. The SBB Activity Context Interface interface declares the shareable attributes of the SBB (see Section 7.4). The SBB Activity Context Interface interface provides a type-safe mechanism for accessing shareable attributes stored in Activity Contexts.

2.2.15 Profile, Profile Table, and Profile Specification

A Profile contains provisioned data. The schema of the Profile defines the attributes that may be provisioned in the Profile. A Profile Table contains zero or more Profiles that conform to the same schema. A Profile Specification defines the interfaces, classes, and deployment descriptor elements needed to define a Profile schema and the interfaces used to provision and access a Profile that conforms to the Profile schema. Zero or more Profile Tables can have the same Profile Specification.

The SLEE specification includes the following SLEE-defined Profile Specifications.

- A Resource Info Profile Specification. *(Deprecated in 1.1)*
  This specifies the schema for all Resource Info Profile Tables and the interfaces used to provision and access a Profile in one of these Profile Tables.
- A generic Address Profile Specification.
  This specifies the schema for all generic Address Profile Tables and the interfaces used to provision and access a Profile in one of these generic Address Profile Tables.

In addition to these SLEE-defined Profile Specifications, SBB Developers can define custom Profile Specifications.

2.2.16 Service

A *Service* is a deployment and management artifact. It specifies a child relation from the SLEE (as the logical parent of all root SBBs, see Figure 1) to a root SBB. Hence, it identifies the root SBB of the Service, the default event delivery priority of the Service, and provides other information needed by the SLEE to instantiate root SBB entities of the Service.

The SLEE will assign the default event delivery priority of the child relation to root SBB entities created by the SLEE for this Service (see Figure 2).

To define a Service, the Administrator provides a Service deployment descriptor element that provides the following:

- A globally unique name of the Service.
- A reference to a root SBB.
- An optional reference to an Address Profile Table.
  An Address Profile Table contains provisioned address data used by the SLEE to determine when new root SBB entities of the Service are created.
- An optional reference to a Resource Info Profile Table.
  Deprecated in 1.1: The Resource Info Profile Table has been deprecated. The Resource Adaptor architecture defined in the SLEE 1.1 specification describes how resource adaptors may interact with Profile Tables and Profiles. SBB Developers should avoid the use of the Resource Info Profile Table as it may be removed in a future version of this specification.
  A Resource Info Profile Table contains provisioned data that the SLEE passes to resource adaptor entities on behalf of the Services that reference this Profile Table.

Two Services may reference the same root SBB, but the Services may provide different event dispatch priorities, reference different Address Profile Tables and Resource Info Profile Tables..

2.2.17 Service state

A Service can be in one of the following three operational states.

- Inactive.
  The Service has been installed successfully and is ready to be activated. The Service is not running, i.e. root SBB entities of the Service's root SBB will not be created to process events.
- Active.
  The Service has been activated, i.e. it is running. The SLEE will create root SBB entities of the Service's root SBB to receive initial events and invoke SBB entities in the SBB entity trees of the Service.
- Stopping.
  The Service is being deactivated. However, some SBB entity trees of the Service still exist in the SLEE and have not completed their processing. The SLEE is waiting for the SBB entities in these SBB entity trees to complete processing so that they can be reclaimed. An SBB entity has completed processing and can be reclaimed when it and all of its child SBB entities are no longer attached to any Activity Context (see Section 7.3.3).

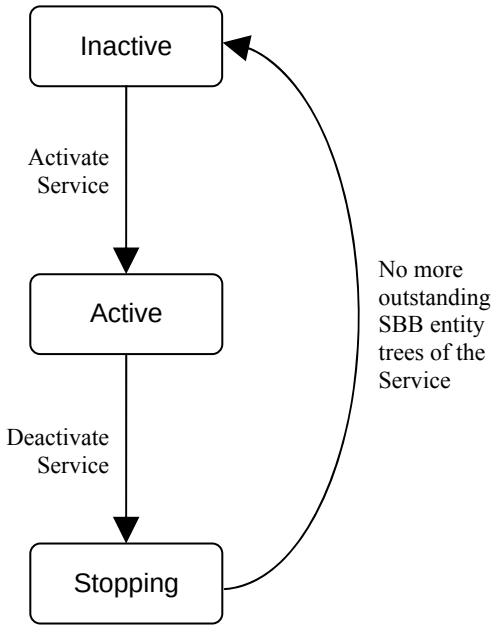

Figure 4 Operational states of a Service

The following steps describe the life cycle of a Service:

- A Service enters the Inactive state when the Service is installed successfully into the SLEE.
- A Service enters the Active state from the Inactive state when the Service is activated. At this point, the SLEE may start a Service Activity for the Service, and fire a Service Started Event on this Activity, as described in Section 8.8.
- A Service enters the Stopping state from the Active state when the Service is deactivated. At this point, the SLEE ends the Service Activity associated with the Service, if it exists (see Section 8.8.2), and fires an Activity End Event on this Activity. SBB entities belonging to the Service that require clean-up when the Service is deactivated should listen to this event and terminate their processing quickly but gracefully when this event is received. Optionally, after some SLEE implementation determined time, the SLEE may also forcefully remove the outstanding SBB entity trees of the Service.
- The SLEE moves a Service to the Inactive state from the Stopping state spontaneously when all outstanding SBB entity trees of the Service complete their processing.

The operational state of a Service is persistent, i.e. the SLEE remembers the last state the Service is in. If the SLEE is shut down and then restarted, the SLEE restores these Services to their previous operational state.

2.2.18 Deployable unit

A deployable unit is a Jar file that can be installed in the SLEE. A deployable unit may contain Services, SBB jar files, event jar files, Profile Specification jar files, resource adaptor type jar files, resource adaptor jar files, and library jar files. Each of these jar files contain the Java class files and the deployment descriptors of one or more of these components.

2.2.19 Management interface

The SLEE specification defines the external management interfaces that may be used by Administrators to manage the SLEE and the Services running in the SLEE. These management interfaces define management

operations. Examples of management operations available through the SLEE's management interfaces include:

- Start and stop the SLEE.
- Install and uninstall a deployable unit.
- Start and stop a Service.
- Create and delete a Profile Table.
- Add, remove, access and modify a Profile in a Profile Table.
- Register and deregister for receipt of alarm notifications.
- Register and deregister for receipt of trace notifications.
- Obtain and reset usage counters and statistics.

2.3 Roles

The SLEE architecture identifies the following roles:

- SBB Developer.
- Service Deployer.
- Resource Adaptor Developer. *(Added in 1.1)*
- Resource Adaptor Deployer. *(Added in 1.1)*
- Profile Specification Developer. *(Added in 1.1)*
- Administrator.

2.3.1 SBB Developer

The SBB Developer designs and implements SBBs. For each SBB, the SBB Developer defines the SBB's child relations, implements the SBB's Java classes, and provides an SBB deployment descriptor. For each child relation, the SBB Developer must provide the default event delivery priority and the child SBB. The SBB Developer packages these files into an SBB jar file.

2.3.2 Service Deployer

The Service Deployer understands the operational environment. The Service Deployer should know which Services have been or will be deployed into the SLEE and the external resources that are plugged into the SLEE.

The Service Deployer creates a deployment descriptor element for each Service. In addition, the Service specifies the default event delivery priority of the Service's root SBB. The Service Deployer may also have to modify the deployment descriptors of the SBBs that are directly or indirectly referenced by the root SBB to adapt these SBBs to the operational environment. For example, the Service Deployer may have to identify and alias Activity Context attributes to resolve ambiguity and conflicts (see Section 7.8), add environment entries, add resource adaptor type, and resource adaptor entity bindings to SBB deployment descriptors.

2.3.3 Resource Adaptor Developer

Added in 1.1.

The Resource Adaptor Developer designs and implements resource adaptors. For each resource adaptor the Resource Adaptor Developer defines the configurable properties of the resource adaptor, implements the Java classes to realize the resource adaptor types implemented by the resource adaptor, and provides a resource adaptor deployment descriptor.

A Resource Adaptor developer may also be involved with the design of resource adaptor type APIs.

2.3.4 Resource Adaptor Deployer

Added in 1.1.

The Resource Adaptor Deployer understands the operational environment that a resource adaptor will be deployed into. The Resource Adaptor Deployer installs resource adaptors and creates resource adaptor entities from those resource adaptors. The Resource Adaptor Deployer configures each resource adaptor entity with appropriate configuration properties so that the resource adaptor entities may successfully interact with the resources they represent.

2.3.5 Profile Specification Developer

Added in 1.1.

The Profile Specification Developer designs the schema of SLEE profile tables and implements the profile CMP, component, and management interfaces that allow services, resource adaptors, and/or the Administrator to interact with profiles. The Profile Specification Developer may also define a set of static query functions that can provide complex profile table search criteria.

An SBB Developer or Resource Adaptor Developer is also often a Profile Specification Developer, however this is not always the case.

2.3.6 Profile Specification Deployer

Added in 1.1.

The Profile Specification Deployer understands the operational environment that a Profile Specification will be deployed into. The Profile Specification Deployer installs profile specification and creates profile tables from those profile specifications. The Profile Specification Deployer may edit the enviroment entries before deployment, and populate profile tables with profiles.

2.3.7 Administrator

The Administrator manages the SLEE and the applications running in the SLEE through the SLEE's management interfaces. An Administrator may be a management client, such as a network management application, network management console, or a person using a management client to manage the SLEE.

Chapter 3 Putting it Together

This chapter describes:

- How an SBB Developer creates SBBs and composes SBBs (see Section 3.1).
- How an SBB Developer creates custom event types (Section 3.2).
- How a Profile Specification Developer creates Profile Specifications (see Section 3.3).
- How a Service Deployer defines a Service and customizes SBBs for deployment (see Section 3.4).
- How an Administrator manages Services and Profiles (see Section 3.5).

3.1 Create an SBB component

Changed in 1.1: The SBB details described below follow the SLEE 1.1 contract for SBBs. For a similar description using the SLEE 1.0 contract for SBBs, the reader is referred to the SLEE 1.0 specification. A 1.1 compliant SLEE must also implement the 1.0 SBB contract. A 1.1 SLEE must use the DTD referenced by an SBB's deployment descriptor to determine whether the SBB follows the 1.0 or 1.1 contract. The SLEE 1.1 specification encourages Service Developers to use the 1.1 DTDs in preference to the 1.0 DTDs as this backward compatibility requirement may be removed in a future version of the SLEE specification.

An SBB Developer creates an SBB from the following parts:

- Zero or more child SBBs, optional (see Section 3.1.2).
- An SBB local interface, optional (see Section 3.1.3).
- An SBB abstract class (see Section 3.1.4).
- An SBB Activity Context Interface interface, optional (see Section 3.1.5).
- An SBB deployment descriptor (see Section 3.1.8).

In addition, the SBB Developer may also define the components needed by the SBB. These components include custom event types (see Section 3.2) and the provisioned data (see Section 3.3) used by the SBB.

3.1.1 Component identity of an SBB component

An SBB's component identity uniquely identifies an SBB component. An SBB's component identity does not identify any particular SBB entity.

An SBB's name, vendor, and version, as defined by the `sbb-name`, `sbb-vendor`, and `sbb-version` elements in the SBB's deployment descriptor specify the component identity of an SBB.

The `SbbID` class (see Section 14.6.5.2) defines the interface of a Java object that encapsulates an SBB's component identity. This Java object is also known as an SBB component identifier. *(Changed in 1.1: SbbID interface replaced by SbbID class.)*

3.1.2 Building an SBB from child SBBs

The SBB Developer can develop and compose an SBB from zero or more child SBBs. The SBB Developer should have a very good understanding of the child SBBs. The SBB Developer should understand each SBB's behavior and how each SBB interacts with other entities via local object invocations, events and Activity Context attributes. These entities include other SBBs, resources, and the SLEE Facilities.

During this process, the SBB Developer:

- Must identify the child relations of the SBB and specify the default event delivery priority of the relation.
- May alias Activity Context attributes.
  Each SBB Activity Context Interface interface specifies the attributes that the SBB is willing to

share with other SBBs. By default, these attributes can be shared among SBB entities of the same SBB component. However, they are not accessible by SBB entities of other SBBs to avoid unintentional sharing resulting from two SBBs developed independently selecting the same attribute name for two semantically distinct attributes. The SBB Developer identifies which attributes declared in child SBB Activity Context Interface interfaces should be aliased. Attribute aliasing instructs the SLEE to make the aliased attributes behave logically as a single attribute. This logical attribute can be updated through any of the aliased attributes' set accessor methods. Changes to the logical attribute are observable through any of the aliased attributes' get accessor methods.

3.1.3 SBB local interface

The SBB Developer may optionally provide an SBB local interface. An SBB entity can invoke another SBB entity synchronously through the SBB entity's SBB local interface (see Chapter 5). The SBB local interface must extend the `javax.slee.SbbLocalObject` interface. The SBB Developer declares the following methods in the SBB local interface:

- Local interface methods.
  The SBB Developer declares each method that may be invoked synchronously by other SBB entities.

This interface is optional. If the SBB Developer does not specify an SBB local interface, the SLEE supplies the generic `SbbLocalObject` interface. This `SbbLocalObject` interface declares the `remove` method used by an SBB entity to remove another SBB entity.

3.1.4 SBB abstract class

The SBB abstract class is mandatory. It contains the processing logic of the SBB. It implements the `javax.slee.Sbb` interface. The SBB Developer provides the following methods:

- Event handler methods (see Section 8.5.2).
  The SBB Developer implements an event handler method for each event type that may be received by the SBB. Each event handler method contains the application logic for processing events of a specific event type.
- Local interface methods (see Section 5.6).
  The SBB Developer implements a method in the SBB abstract class for each method declared in the SBB local interface.
- Life cycle callback methods (see Section 6.3).
  An SBB object has a life cycle. The SLEE invokes one of these life cycle callback methods when an SBB object transitions from one life cycle state to another life cycle state (see Section 6.2).
- An `sbbExceptionThrown` callback method (see Section 6.9.3).
  The SLEE invokes this method after a SLEE originated invocation of a transactional method of the SBB object returns by throwing a `RuntimeException`.
- An `sbbRolledBack` callback method (see Section 6.10).
  The SLEE invokes this method after a transaction that was used to invoke a transactional method of the SBB abstract class has been rolled back.
- Initial event selector methods (see Section 8.6.4).
  The SBB Developer implements these optional methods to provide custom ways to determine if an event is an initial event of the SBB.

The SBB Developer declares the following abstract methods. The SLEE implements these abstract methods when the SBB is deployed.

- Fire event methods (see Section 8.5.1).
  The SBB Developer declares an abstract method for each event type that may be fired by the SBB.
- CMP field get and set accessor methods (see Section 6.5).
  The SBB Developer declares an abstract get and an abstract set method for each CMP field.

- Get child relation methods (see Section 6.8).
  The SBB Developer declares a get child relation method for each child relation of the SBB. The object returned by a get child relation method has methods to access the child SBB entities of the relation and create additional SBB entities for this relation.
- Get Profile CMP methods (see Section 6.7).
  Deprecated in 1.1: Get Profile CMP methods have been deprecated. SBB Developers are encouraged to use the new Profile Table (see Section 10.8) and Profile Local interfaces (see Section 10.7).
  The SBB Developer declares an abstract get Profile CMP method for each Profile CMP interface (i.e. Profile Specification) accessed by the SBB.
- Get SBB Usage Parameter methods (see Section 11.4.1.1).
  The SBB Developer may declare abstract get SBB Usage Parameter methods used by the SBB to access objects that implement the SBB Usage Parameters interface of the SBB. Each of these objects represents a set of usage parameters that the SBB may manipulate (see Section 11.3).
- SBB Activity Context Interface narrow method (see Section 7.7.2).
  The SBB Developer declares a method used to narrow an object that implements the generic `ActivityContextInterface` interface to an object that implements the SBB Activity Context Interface interface of the SBB. This method is optional. The SBB Developer does not declare this method if the SBB Developer does not define an SBB Activity Context Interface interface.

3.1.5 SBB Activity Context Interface interface

The SBB Developer may provide an optional SBB Activity Context Interface interface, which defines the SBB component specific view of an Activity Context.. The SBB uses this interface to access shared attributes stored in the Activity Context. The SBB Activity Context Interface interface must extend the generic `ActivityContextInterface` interface. It declares the shareable state of the SBB as a set of attributes (see Section 7.5).

If the SBB Developer provides an SBB Activity Context Interface interface, the SBB Developer declares the following abstract methods in the SBB Activity Context Interface interface:

- Get and set accessor methods.
  The SBB Developer declares get and set accessor methods for each Activity Context attribute.

This interface is optional. If the SBB does not have any shareable attributes, the SBB Developer does not have to provide an SBB Activity Context Interface interface.

If the SBB does not define an SBB Activity Context Interface interface, it interacts with Activity Contexts through objects that implement the generic `ActivityContextInterface` interface that is defined by the SLEE specification. The generic `ActivityContextInterface` interface defines a method that provides access to the underlying Activity object of an Activity Context.

If the SBB Developer declares an SBB Activity Context Interface interface, the SBB Developer must also declare an SBB Activity Context Interface interface narrow method to get access to objects that implement the SBB Activity Context Interface interface. *(Clarified in 1.1)*

3.1.6 SBB Usage Parameters interface

The SBB Developer may optionally provide an SBB Usage Parameters interface. The SBB declares the names and types of the usage parameters relevant to this SBB in the SBB Usage Parameters interface by declaring the methods used to manipulate these usage parameters. There are two types of usage parameters: counter-type and sample-type. An SBB can increment and decrement a counter-type usage parameter, while a sample-type usage parameter accumulates data samples.

If the SBB Developer provides an SBB Usage Parameters interface, the SBB Developer declares the following abstract methods in the SBB Usage Parameters interface:

- Increment methods.
  The SBB Developer declares increment methods to declare the presence of and to update counter-type usage parameters. The SLEE derives the name of each usage parameter from the name of each increment method.
- Sample methods.
  The SBB Developer declares sample method to declare the presence of and to add samples to sample-type usage parameters. The SLEE derives the name of each usage parameter from the name of each sample method.

This interface is optional. If the SBB does not require any usage parameters, the SBB Developer does not have to provide an SBB Usage Parameters interface.

If the SBB Developer declares an SBB Usage Parameters interface, the SBB Developer must also declare (in the SBB abstract class) at least one of the get SBB Usage Parameter methods to get access to objects that implement the SBB Usage Parameters interface.

3.1.7 Recommended SBB interfaces and classes naming convention

The SLEE specification recommends the following naming convention for the interfaces and classes of an SBB component.

Class or interface	*Recommended Naming Convention*
SBB abstract class	<*name*>`Sbb`
SBB Activity Context Interface interface	<*name*>`SbbActivityContextInterface`
SBB local interface	<*name*>`SbbLocalObject`

3.1.8 SBB deployment descriptor

The SBB Developer identifies the SBB's interfaces, classes, child relations, and Activity Context attribute aliases in an SBB deployment descriptor. References from the SBB to other components are also included in the deployment descriptor. These other components include libraries, profile specifications, resource adaptor types, events and other SBB components. The deployment descriptor also allows SBB component environment entries to be set.

An SBB deployment descriptor contains an `sbb` element. The `sbb` element contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- An `sbb-name` element, an `sbb-vendor` element, and an `sbb-version` element.
  These elements identify the SBB component.
- An `sbb-alias` element.
  This element is optional. It assigns an alias to the SBB. Other elements in the same `sbb` element use this alias to reference the SBB. It allows other elements in the enclosing `sbb` element to refer to the SBB without specifying the SBB's name, vendor and version. This element may be omitted if the SBB is not referenced by other elements (such as `get-child-relation-method` elements) in the same `sbb` element. The scope of this alias is the enclosing `sbb` element.
- Zero or more `library-ref` elements. *(Added in 1.1).*
  Each library component required by the SBB of the enclosing `sbb` element must be identified by a `library-ref` element. A `library-ref` element contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - A `library-name` element, a `library-vendor` element, and a `library-version` element.
    These elements uniquely identify the referenced library component and must match ex-

actly the corresponding `library-name`, `library-vendor`, and `library- version` elements specified in the referenced library's `library` element.

- Zero or more `sbb-ref` elements.
  Each SBB referenced by the SBB of the enclosing `sbb` element must be identified by an `sbb-ref` element. An `sbb-ref` element contains the following sub-elements.
  - A `description` element.
    This is an optional informational element.
  - An `sbb-name` element, an `sbb-vendor` element, and an `sbb-version` element.
    These elements uniquely identify the referenced SBB component and must match exactly the corresponding `sbb-name`, `sbb-vendor`, and `sbb-version` elements specified in the referenced SBB's `sbb` element.
  - An `sbb-alias` element.
    This element assigns an alias to the referenced SBB. It allows other elements in the enclosing `sbb` element to refer to the SBB without specifying the SBB's name, vendor and version. Other elements in the enclosing `sbb` element use this alias to reference the aliased SBB. The scope of this alias is the enclosing `sbb` element.
- Zero or more `profile-spec-ref` elements.
  Each Profile Specification used by the SBB must be identified by a `profile-spec-ref` element. A `profile-spec-ref` element contains the following sub-elements.
  - A `description` element.
    This is an optional informational element.
  - A `profile-spec-name` element, a `profile-spec-vendor` element, and a `profile-spec-version` element.
    These elements uniquely identify the Profile Specification, and must match exactly the corresponding `profile-spec-name`, `profile-spec-vendor`, and `profile-version` elements specified in the referenced Profile Specification's `profile-spec` element.
  - A `profile-spec-alias` element.
    Deprecated in 1.1: Since get Profile CMP method has been deprecated, this element is also deprecated.
    This element is optional. It assigns an alias to the Profile Specification. It allows other elements in the enclosing `sbb` element to refer to the referenced Profile Specification without specifying the Profile Specification's name, vendor and version. Other elements in the enclosing `sbb` element use this alias to reference the Profile Specification. The scope of this alias is the enclosing `sbb` element.
    Note: In the SLEE 1.0 specification the `profile-spec-alias` element was mandatory. This element is only required in the SLEE 1.1 specification if the SBB Developer continues to define a get Profile CMP method in their SBB for the referenced Profile Specification, rather than interacting with Profiles of the referenced Profile Specification via Profile Table and Profile Local interfaces.
- An `sbb-classes` element.
  This element is mandatory. An `sbb-classes` element contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - An `sbb-abstract-class` element.
    It contains the following attribute and sub-elements:

- A `reentrant` attribute.
  This boolean attribute indicates whether the SBB component is re-entrant (see Section 6.11). If this attribute is not specified, the default value is "False".
- A `description` element.
  This is an optional informational element.
- An `sbb-abstract-class-name` element.
  This element identifies the class name of the SBB abstract class.
- Zero or more `cmp-field` elements.
  Each CMP field defined in the SBB abstract class must be identified by a `cmp-field` element. Each `cmp-field` element contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - A `cmp-field-name` element.
    This element identifies the name of the CMP field (see Section 6.5).
  - An `sbb-alias-ref` element.
    This optional element references an SBB by its `sbb-alias` that is specified within the same `sbb` element. When this element is present, only `null` and references to SBB entities of the SBB identified by this element can be stored in the CMP field. *(Changed in 1.1: It is now optional for CMP fields that store SBB local interface objects (i.e. the CMP field type is `javax.slee.SbbLocalObject` or a type derived from `SbbLocalObject`) to define this element.)*
- Zero or more `get-child-relation-method` elements.
  Each get child relation method declared in the SBB abstract class must be identified by a `get-child-relation-method` element. A get child relation method declares a parent-child relation from the SBB to the SBB identified by the enclosed `sbb-alias-ref` element. Each `get-child-relation-method` element has the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - An `sbb-alias-ref` element.
    This element references an SBB by its `sbb-alias` that is specified within the same `sbb` element.
  - A `get-child-relation-method-name` element.
    This element specifies the method name of the get child relation method in the SBB abstract class. The SBB entity uses this method to get an object that can be invoked to create child SBB entities and to enumerate child SBB entities of this relation.
  - A `default-priority` element.
    This element is mandatory. A `default-priority` element specifies the default event delivery priority of the child SBB relative to its sibling SBBs (see Section 8.6.7).
- Zero or more `get-profile-cmp-method` elements.
  Deprecated in 1.1: Get Profile CMP methods have been deprecated. SBB Developers are encouraged to use the new Profile Table (see Section 10.8) and Profile Local interfaces (see Section 10.7).
  Each get Profile CMP method declared in the SBB abstract class must be identi-

fied by a `get-profile-cmp-method` element. Each `get-profile-cmp-method` element has the following sub-elements:

- A `description` element.
  This is an optional informational element.
- A `profile-spec-alias-ref` element.
  This element references a Profile Specification by its `profile-spec-alias` that is specified in a `profile-spec` element within the same `sbb` element.
- A `get-profile-cmp-method-name` element.
  This element specifies the name of the method used to get an object that implements a Profile CMP interface from a Profile identifier. The Profile CMP interface is specified by the Profile Specification identified by the above `profile-spec-alias-ref` element.

- An `sbb-local-interface` element.
  This element is optional. If the SBB developer defines an SBB local interface, this element identifies the SBB local interface. It contains the following attributes and sub-elements:
  - An `isolate-security-permissions` attribute. *(Added in 1.1)*
    This attribute controls whether or not security permissions of other protection domains in the call stack are propagated to the SBB when a business method on the SBB Local interface is invoked. If the value of this attribute is "False", then the method in the SBB abstract class invoked as a result of a business method invoked on the SBB Local interface runs with an access control context that includes the protection domain(s) of the SBB as well as the protection domains of any other classes in the call stack as prescribed by the Java security model, such as the SBB that invoked the SBB Local interface method. If the value of this attribute is "True", the SLEE automatically wraps the method invoked on the SBB abstract class in response to the SBB Local interface method invocation in an `AccessController.doPrivileged` block in order to isolate the security permissions of the invoked SBB, i.e. the security permissions of other protection domains in the call stack do not affect the invoked SBB.
  - A `description` element.
    This is an optional informational element.
  - An `sbb-local-interface-name` element.
    This element identifies the class name of the SBB local interface.
- An `sbb-activity-context-interface` element.
  This element is optional. If the SBB developer defines an SBB Activity Context Interface interface, this element identifies the SBB Activity Context Interface interface. It contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - An `sbb-activity-context-interface-name` element.
    This element identifies the class name of the SBB Activity Context Interface interface.
- An `sbb-usage-parameters-interface` element.
  This element is optional. If the SBB developer defines an SBB Usage Parameters interface, this element identifies the SBB Usage Parameters interface. It contains the following sub-elements:

        - A `description` element.
          This is an optional informational element.
        - An `sbb-usage-parameters-interface-name` element.
          This element identifies the class name of the SBB Usage Parameters interface.

- An `address-profile-spec-alias-ref` element.
  This element is optional. If the SBB can be a root SBB, this element identifies the Address Profile Specification of the SBB. This element references a Profile Specification by its `profile-spec-alias` that is specified in a `profile-spec-ref` element in the same `sbb` element.
- Zero or more `event` elements.
  Each event type that the SBB may receive or fire must have an `event` element (see Section 8.1). An `event` element contains the following attributes and sub-elements:
  - An `initial-event` attribute.
    This attribute indicates if the event type is an initial event type (see Section 8.6.1).
  - An `event-direction` attribute.
    This attribute indicates if the SBB receives or fires the event type identified by the `event-type-ref` element of the `event` element. The value of this attribute must be "Receive", "Fire", or "FireAndReceive". If `initial-event` is true, then `event-direction` must be either "Receive" or "FireAndReceive".
  - A `mask-on-attach` attribute.
    This attribute indicates whether events of the event type identified by the `event-type-ref` element should be masked when an SBB entity attaches to an Activity Context (see Section 7.4.2).
  - A `last-in-transaction` attribute.
    This optional attribute indicates whether the event handler method is the last event handler method invoked in a transaction. This attribute is only applicable to `event` elements where the `event-direction` attribute is set to "Receive" or "FireAndReceive". If not specified its default value is "True". For more information on this attribute refer to section 9.8.4.
  - A `description` element.
    This is an optional informational element.
  - An `event-type-ref` element.
    This element references an event type. It contains the following elements:
    - An `event-type-name` element, an `event-type-vendor` element, and an `event-type-version` element.
      These elements uniquely identify an event type declared in an `event-definition` element specified in another deployment descriptor. An `event-definition` element declares an event type (see Section 8.4.1)
  - An `event-name` element.
    This element provides the SBB scoped name that the SBB uses to refer to the event type identified by the `event-type-ref` element (see Section 8.1.10).
  - Zero or more `initial-event-select` elements.
    These elements are only meaningful if `initial-event` is true. They indicate which convergence name variables should be selected (see Section 8.6.3). It has the following attribute.
    - A `variable` attribute.
      The value of this attribute can be one of "ActivityContext", "AddressProfile", "Address", "EventType", or "Event".

  - An `initial-event-selector-method-name` element.
    This element is optional and is meaningful only if `initial-event` is true. It identifies an initial event selector method. The SLEE invokes this optional method to determine if an event of the specified event type is an initial event if the SBB is a root SBB (see Section 8.6.4).
  - An `event-resource-option` element.
    This element is optional. It provides additional event handling options that the SLEE forwards to resource adaptor entities that emit events of the event type identified by the `event-type-ref` element (see Section 8.4.2).

- Zero or more `activity-context-attribute-alias` elements.
  These elements are optional. These elements and Activity Context attribute aliasing are described in detail in Section 7.8. Each of these elements contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - An `attribute-alias-name` element.
    This element specifies an alias name. The alias name logically identifies the aliased logical attribute. This name is globally scoped. If two or more `activity-context-attribute-alias` elements specify the same `attribute-alias-name`, all the attributes specified by their `sbb-activity-context-attribute` elements are aliased.
  - One or more `sbb-activity-context-attribute-name` elements.
    Each element identifies an attribute declared in the SBB Activity Context Interface interface of the SBB that should be aliased. Each SBB Activity Context Interface attribute can be aliased at most once, i.e. it is either not aliased or aliased through a single alias name.
- Zero or more `env-entry` elements.
  These elements are optional. Each `env-entry` element binds an environment entry into the JNDI component environment of the SBB (see Section 6.13). Each `env-entry` element contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - An `env-entry–name` element.
    This element specifies the location within the JNDI component environment to which the environment entry value will be bound.
  - An `env-entry–type` element.
    This element specifies the type of the value specified by the environment `env-entry` element.
  - An `env-entry-value` element.
    This element specifies the value that will be bound to the location specified by the `env-entry–name` element.
- Zero or more `resource-adaptor-type-binding` elements.
  These elements are optional. Each `resource-adaptor-type-binding` element declares a resource adaptor type that the SBB depends on (see `resource-adaptor-type-ref` below). It may optionally bind an object that implements the resource adaptor type defined Activity Context Interface Factory interface into the JNDI component environment of the SBB (see Section 6.13.2). If necessary, it may also contain `resource-adaptor-entity-binding` sub-elements that binds resource adaptor entities of the same resource adaptor type to be SBB. Each `resource-adaptor-type-binding` element contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- A `resource-adaptor-type–ref` element.
  This element identifies the resource adaptor type that defines interface of the Activity Context Interface Factory object. It contains the following sub-elements:
  - A `resource-adaptor-type-name` element, a `resource-adaptor-type-vendor` element, and a `resource-adaptor-type-version` element.
    These elements uniquely identify the resource adaptor type, and must match exactly the corresponding `resource-adaptor-type-name`, `resource-adaptor-type-vendor`, and `resource-adaptor-type-version` elements specified in the resource adaptor type's `resource-adaptor-type` element.
- An `activity–context-interface-factory-name` element.
  This optional element specifies the location within the JNDI component environment to which the Activity Context Interface Factory object will be bound.
- Zero or more `resource-adaptor-entity-binding` elements.
  Each `resource-adaptor-entity-binding` element binds an object that implements the resource adaptor interface of the resource adaptor type into the JNDI component environment of the SBB (see Section 6.13.3). Each `resource-adaptor-entity-binding` element contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - A `resource-adaptor-object–name` element.
    This element specifies the location within the JNDI component environment to which the object that implements the resource adaptor interface will be bound.
  - A `resource-adaptor-entity-link` element.
    This is an optional element. It identifies the resource adaptor entity that provides the object that should be bound into the JNDI component environment of the SBB. The name specified in the `resource-adaptor-entity-link` element must be equal to a link name that a resource adaptor entity has been bound to using the `bindLinkName` method of the `ResourceManagementMBean` interface (refer Section 14.12). The identified resource adaptor entity must be an instance of a resource adaptor whose resource adaptor type is specified by the `resource-adaptor-type-ref` sub-element of the enclosing `resource-adaptor-type-binding` element.

- Zero or more `ejb-ref` elements.
  (Changed in 1.1: The `ejb-link` element has been removed. The method in which an EJB of the appropriate type is identified to the SLEE is SLEE-vendor specific.)
  These elements are optional. Each `ejb-ref` element binds a reference to an Enterprise JavaBean home into the JNDI component environment of the SBB. This allows the SBB to invoke Enterprise JavaBeans. Section 20.3 in the "Enterprise JavaBeans 2.0, Final Release" specification describes this element and how components get access to the referenced Enterprise JavaBean home. This element contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - An `ejb-ref-name` element.
    This element specifies the location within the JNDI component environment to which the EJB will be bound.

- An `ejb-ref-type` element.
  This element specifies the expected type of the EJB, either Entity or Session.
- A `home` element.
  This element specifies the expected Java type of the EJB's remote home interface.
- A `remote` element.
  This element specifies the expected Java type of the EJB's remote component interface.

3.1.9 SBB jar file

The SBB jar file is the standard format for packaging one or more SBBs. It must include the following:

- An SBB jar deployment descriptor.
  - The SBB jar deployment descriptor is stored with the name `META-INF/sbb-jar.xml` in the SBB jar file.
  - The root element of the SBB jar deployment descriptor is an `sbb-jar` element. This element has the following sub-elements.
    - A `description` element.
      This is an optional informational element.
    - One or more `sbb` elements.
      Each of these elements is an SBB's deployment descriptor as defined in Section 3.1.8.
    - A `security-permissions` element.
      This element is optional and contains sub-elements that identify additional security permissions that should be granted to the classes in SBB jar file.
      - A `description` element.
        This is an optional informational element.
      - A `security-permission-spec` element.
        This element identifies the security permission policies used by the SBB jar file classes. The syntax of this element is defined at the following URL:
        `http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html#FileSyntax`.
        Note: If the `codeBase` argument is not specified for a `grant` entry in the provided security permission specification, the SLEE assumes the code base to be the SBB jar file, and the security permissions specified in the `grant` entry are granted to all classes loaded from the SBB jar file, i.e. they apply to all SBBs defined in the SBB jar file. The security permissions are not granted to classes loaded from any other dependent component jar required by the SBBs defined in the SBB deployment descriptor. If the `codeBase` argument is specified for a `grant` entry, the specified path is taken to be relative to the root directory of the SBB jar within the deployable unit jar, but its use is otherwise undefined by the SLEE specification.
- Class files of the SBBs specified by the `sbb` elements of the `sbb-jar` element.
  - The SBB jar file must contain, either by inclusion or by reference, the class files of each SBB.
  - The SBB jar file must also contain, either by inclusion or by reference, the class files for all the classes and interfaces that the SBBs' classes and interfaces depend on, except J2SE classes, SLEE classes, classes included in the SBB jar files of referenced SBBs of the SBBs (see `sbb-ref` element), classes included in the event jar files of the event

types referenced by the SBBs (through `event-type-ref` elements), classes included in the Profile Specification jar files of the Profile Specifications referenced by the SBBs (through `profile-spec-ref` elements), classes included in the resource adaptor type jar files referenced by the SBBs (through `resource-adaptor-type-binding` elements) , or classes included in the library jar files of the libraries referenced by the SBBs (through the `library-ref` elements). This includes their superclasses and superinterfaces, dependent classes, and the classes and interfaces used as method parameters, results, and exceptions.

A jar file contains a second file "by reference", if the second file is named in the Class-Path attribute in the Manifest file of the referencing jar file or is contained (either by inclusion or by reference) in another jar file that is named in the Class-Path attribute in the Manifest file of the referencing jar file. However, the SLEE specification strongly discourages use of the Manifest Class-Path attribute as its meaning and use in SLEE is poorly defined. The SLEE specification instead recommends the use of the clearly-defined library component jars as containers of common library code.

3.1.10 SBB jar file example

The following example illustrates an SBB jar file that contains a Foo SBB, and a Bar SBB. The Bar SBB is a parent SBB of the Foo SBB. This example also shows the naming convention described in Section 3.1.7 and the deployment descriptors of the Foo SBB and Bar SBB.

The deployment descriptor for the example SBB jar file is as follows:

```
<sbb-jar>
        <sbb>
                <description> FooSbb … </description>
                <sbb-name> Foo SBB </sbb-name>
                <sbb-vendor> com.foobar </sbb-vendor>
                <sbb-version> 1.0 </sbb-version>
                …
                <profile-spec-ref> … </profile-spec-ref>
                …
                <sbb-classes>
                        <sbb-abstract-class>
                                <description> … </description>
                                <sbb-abstract-class-name>
                                        com.foobar.FooSbb
                                </sbb-abstract-class-name>
                                <cmp-field> … </cmp-field>
                                …
                        </sbb-abstract-class>
                        <sbb-local-interface>
                                <description> … </description>
                                <sbb-local-interface-name>
                                        com.foobar.FooSbbLocalObject
                                </sbb-local-interface-name>
                        </sbb-local-interface>
                        <sbb-activity-context-interface>
                                <description> … </description>
                                <sbb-activity-context-interface-name>
                                        com.foobar.FooSbbActivityContextInterface
                                </sbb-activity-context-interface-name>
                        </sbb-activity-context-interface>
                        <sbb-usage-parameter-interface>
                                <description> … </description>
                                <sbb-usage-parameter-interface-name>
                                        com.foobar.FooSbbUsageParameter
                                </sbb-usage-parameter-interface-name>
                        </sbb-usage-parameter-interface>
                </sbb-classes>
                …
                <event> … </event>
                …
```

```
                <activity-context-attribute-alias> … </activity-context-attribute-alias>
                …
                <env-entry> … </env-entry>
                …
                <resource-adaptor-type-binding>
                        …
                        <resource-adaptor-entity-binding>
                                …
                        </resource-adaptor-entity-binding>
                        …
                </resource-adaptor-type-binding>
        </sbb>
        <sbb>
                <description> BarSbb … </description>
                <sbb-name> Bar SBB </sbb-name>
                <sbb-vendor> com.foobar </sbb-vendor>
                <sbb-version> 1.0 </sbb-version>
                …
                <sbb-ref>
                        <description> … </description>
                        <sbb-name> Foo SBB </sbb-name>
                        <sbb-vendor> com.foobar </sbb-vendor>
                        <sbb-version> 1.0 </sbb-version>
                        <sbb-alias> FooSBB </sbb-alias>
                </sbb-ref>
                …
                <sbb-classes>
                        <sbb-abstract-class>
                                …
                                <sbb-abstract-class-name>
                                        com.foobar.BarSbb
                                </sbb-abstract-class-name>
                                …
                                <get-child-relation-method>
                                        <sbb-alias-ref> FooSBB </sbb-alias-ref>
                                        <get-child-relation-method-name>
                                                …
                                        </get-child-relation-method-name>
                                        <default-priority> … </default-priority>
                                </get-child-relation-method >
                                …
                        </sbb-abstract-class>
                </sbb-classes>
                …
        </sbb>
</sbb-jar>
```

The content of the SBB jar file is as follows:

```
META-INF/sbb-jar.xml
META-INF/MANIFEST.MF
…
com/foobar/BarSbb.class
com/foobar/FooSbb.class
com/foobar/FooSbbLocalObject.class
com/foobar/FooSbbActivityContextInterface.class
com/foobar/FooSbbUsageParameter.class
…
```

3.2 Custom event types

Custom event types are event types defined by an SBB Developer for collaborating with other SBBs in the same or other services. For each custom event type, the SBB Developer provides the following parts:

- Custom event class.
  The custom event class defines the methods and data of a custom event object.
- Custom event type.
  A custom event type is defined by the custom event type's name, vendor and version.

- An event definition deployment descriptor.

3.2.1 Event type deployment descriptor

Each custom event type must have an event definition deployment descriptor. An event definition deployment descriptor contains an `event-definition` element. Each `event-definition` element contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- An `event-type-name` element, an `event-type-vendor` element and an `event-version` element.
  These elements uniquely identify the event type.
- An `event-class-name` element.
  This element identifies the Java class associated with the event type.

The SLEE and resource adaptor types also use `event-definition` elements to define the event types they fire.

3.2.2 Event type packaging

The SBB Developer may distribute the event classes in either source form or compiled Java class files. These files and the event definition deployment descriptor files may be distributed as separate files, or in one or more jar files.

To ease distribution and deployable unit assembly, the SLEE specification recommends that the SBB Developer distribute these files in an event jar file. The SLEE specification specifies the organization and content of an event jar file.

3.2.3 Event jar file

The event jar file is the standard format for packaging one or more event types. It must include the following:

- An event jar deployment descriptor.
  - The event jar deployment descriptor is stored with the name `META-INF/event-jar.xml` in the event jar file.
  - The root element of the event jar deployment descriptor is an `event-jar` element. This element contains the following sub-elements.
    - A `description` element.
      This is an optional informational element.
    - Zero or more `library-ref` elements. *(Added in 1.1)*
      Each library component required by the event types defined in the event jar must be identified by a `library-ref` element. A `library-ref` element contains the following sub-elements:
      - A `description` element.
        This is an optional informational element.
      - A `library-name` element, a `library-vendor` element, and a `library-version` element.
        These elements uniquely identify the referenced library component and must match exactly the corresponding `library-name`, `library-vendor`, and `library-version` elements specified in the referenced library's `library` element.

        - One or more `event-definition` elements.
          Each of these elements is an event definition deployment descriptor as defined in Section 3.2.1.

- Class files of the event types specified by the `event-definition` elements of the `event-jar` element.
    - The event jar file must contain, either by inclusion or by reference, the class file of the event class.
    - The event jar file must also contain, either by inclusion or by reference, the class files for all the classes and interfaces that the event class depends on, except J2SE classes and SLEE classes. This includes their superclasses and superinterfaces, dependent classes, and the classes and interfaces used as method parameters, results, and exceptions.

3.2.4 Event jar file example

The following example illustrates an event jar file that contains two custom event types.

The deployment descriptor for the example event jar file is as follows:

```
<event-jar>
        …
        <event-definition>
                …
                <event-type-name> com.foobar.event.HelpRequestedEvent </event-type-name>
                <event-type-vendor> com.foobar </event-type-vendor>
                <event-type-version> 1.0a </event-type-version>
                <event-class-name> com.foobar.event.HelpRequestedEvent </event-class-name>
        </event-definition>
        <event-definition>
                …
                <event-type-name> com.foobar.event.IVRCompletedEvent </event-type-name>
                <event-type-vendor> com.foobar </event-type-vendor>
                <event-type-version> 1.1 </event-type-version>
                <event-class-name> com.foobar.event.IVRCompletedEvent </event-class-name>
        </event-definition>
        …
</event-jar>
```

The content of the event jar file is as follows:

```
META-INF/event-jar.xml
META-INF/MANIFEST.MF
…
com/foobar/event/HelpRequestedEvent.class
com/foobar/event/IVRCompletedEvent.class
…
```

3.3 Profile Specification

Changed in 1.1: The Profile Specification details described here follow the SLEE 1.1 contract for Profiles. For a similar description using the SLEE 1.0 contract for Profiles, the reader is referred to the SLEE 1.0 specification. A 1.1 compliant SLEE must also implement the 1.0 Profile Specification contract. A 1.1 SLEE must use the DTD referenced by a Profile Specification's deployment descriptor to determine whether the Profile Specification follows the 1.0 or 1.1 contract. The SLEE 1.1 specification encourages Profile Specification Developers to use the 1.1 DTDs in preference to the 1.0 DTDs as this backward compatibility requirement may be removed in a future version of the SLEE specification.

For each type of provisioned data used by the SBB, a Profile Specification Developer either provides a new Profile Specification or identifies an existing Profile Specification that may be used by the SBB. For each Profile Specification, the Profile Specification Developer provides the following parts:

- A Profile CMP interface.
- An optional Profile Local interface.

- An optional Profile Management interface.
- An optional Profile abstract class.
- An optional Profile Table interface.
- An optional Profile Usage Parameters interface.
- A Profile Specification deployment descriptor.

3.3.1 Profile CMP interface

This interface defines the schema of a Profile Table created from this Profile Specification, i.e. it defines the attributes of each Profile in the Profile Table. The Profile Specification Developer declares the following methods in the Profile CMP interface:

- Profile attribute get and set accessor methods.
  The Profile Specification Developer declares each attribute using a pair of get and set accessor methods in this interface.

3.3.2 Profile Local interface

Added to 1.1: This allows Profiles to define business methods and allows write access to Profiles by SLEE Components.

The Profile Specification Developer may provide an optional Profile Local interface. This interface declares the methods that should be visible to SLEE Components such as SBBs, Resource Adaptors, or other Profile Specifications.

If the Profile Specification Developer provides a Profile Local interface, the Profile Specification Developer declares the following methods:

- Visible Profile attribute get and set accessor methods.
  Each of these methods may be declared in the Profile Local interface or in a base interface of the Profile Local interface. The Profile Local interface may extend the Profile CMP interface to include all the accessor methods that are declared in the Profile CMP interface in the Profile Local interface.
- Business methods.
  These methods are methods declared in the Profile Local interface that are not also declared in the Profile CMP interface.

If the Profile Specification Developer does not provide a Profile Local interface, then the SLEE provides an anonymous Profile Local interface that makes all methods declared in the Profile CMP interface visible as read-only attribute to SLEE Components.

3.3.3 Profile Management interface

The Profile Specification Developer may provide an optional Profile Management interface. This interface declares the methods that should be visible to a management client. The management client may be a network management console, tool, or an Administrator.

If the Profile Specification Developer provides a Profile Management interface, the Profile Specification Developer declares the following methods:

- Visible Profile attribute get and set accessor methods.
  Each of these methods may be declared in the Profile Management interface or in a base interface of the Profile Management interface. The Profile Management interface may extend the Profile CMP interface to include all the accessor methods that are declared in the Profile CMP interface in the Profile Management interface.
- Management methods.
  These methods are methods declared in the Profile Management interface that are not also declared in the Profile CMP interface.

If the Profile Specification Developer does not provide a Profile Management interface, then the SLEE makes all methods declared in the Profile CMP interface visible to management clients.

3.3.4 Profile abstract class

Changed in 1.1: The Profile abstract class replaces the Profile Management abstract class defined in 1.0.

The Profile abstract class implements the business methods defined in the Profile Local interface, the management methods defined in the Profile Management interface and the generic `javax.slee.profile.Profile` interface. The `Profile` interface defines the Profile life cycle callback methods. The Profile abstract class also implements the Profile CMP interface, but the SLEE implements the get and set accessor methods declared in the Profile CMP interface when the Profile Specification is deployed. The Profile abstract class may also define get Usage Parameters methods to allow Profiles to collect usage information.

If the Profile Specification Developer provides a Profile abstract class, the Profile Specification Developer implements the following methods:

- Business methods declared by the Profile Specification Developer in the Profile Local interface.
- Management methods declared by the Profile Specification Developer in the Profile Management interface.
- Life cycle callback methods declared in the SLEE specification defined `Profile` interface.
  An instance of the Profile abstract class is known as a Profile object, and has its own life cycle. The SLEE invokes one of these life cycle callback methods when a Profile object transitions from one life cycle state to another life cycle state (see Section 10.13.2).

3.3.5 Profile Table interface

Added in 1.1: This interface has been added to allow SLEE components to manipulate Profiles in Profile Tables and execute queries.

The Profile Table interface defines the methods that allow SLEE Components such as SBBs and Resource Adaptors to create, find, and remove Profiles from a Profile Table. The Profile Table interface must extend the SLEE-defined `javax.slee.profile.ProfileTable` interface. The `ProfileTable` interface defines the create, find, and remove methods. The SLEE implements the Profile Table interface when the Profile Specification is deployed.

If the Profile Specification Developer provides a Profile Table interface, the Profile Specification Developer define the following methods:

- Static query methods. These methods are used to invoke static queries on the Profile Table. Static queries are defined in the Profile Specification's deployment descriptor (see Section 10.20.2).

3.3.6 Profile Usage Parameters interface

Added in 1.1: This interface allows Profiles to collect usage information.

The Profile Specification Developer may optionally provide a Profile Usage Parameters interface. The Profile Specification declares the names and types of the usage parameters relevant to the Profiles in the Profile Usage Parameters interface by declaring the methods used to manipulate these usage parameters. There are two types of usage parameters: counter-type and sample-type. A Profile can increment and decrement a counter-type usage parameter, while a sample-type usage parameter accumulates data samples.

If the Profile Specification Developer provides a Profile Usage Parameters interface, the Profile Specification Developer declares the following abstract methods in the Profile Usage Parameters interface:

- Increment methods.
  The Profile Specification Developer declares increment methods to declare the presence of and to update counter-type usage parameters. The SLEE derives the name of each usage parameter from the name of each increment method.

- Sample methods.
  The Profile Specification Developer declares sample method to declare the presence of and to add samples to sample-type usage parameters. The SLEE derives the name of each usage parameter from the name of each sample method.

This interface is optional. If the Profile Specification does not require any usage parameters, the Profile Specification Developer does not have to provide a Profile Usage Parameters interface.

If the Profile Specification Developer declares a Profile Usage Parameters interface, the Profile Specification Developer must also declare (in the Profile abstract class) at least one of the get Usage Parameters methods to get access to objects that implement the Profile Usage Parameters interface.

3.3.7 Profile Specification deployment descriptor

Each Profile Specification must have a Profile Specification deployment descriptor. A Profile Specification deployment descriptor contains a `profile-spec` element. Each `profile-spec` element contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- A `profile-spec-name` element, a `profile-spec-vendor`, and a `profile-spec-version` element.
  These elements uniquely identify a Profile Specification.
- Zero or more `library-ref` elements. *(Added in 1.1).*
  Each library component required by the Profile Specification of the enclosing `profile` element must be identified by a `library-ref` element. A `library-ref` element contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - A `library-name` element, a `library-vendor` element, and a `library-version` element.
    These elements uniquely identify the referenced library component and must match exactly the corresponding `library-name`, `library-vendor`, and `library- version` elements specified in the referenced library's `library` element.
- Zero or more `profile-spec-ref` elements. *(Added in 1.1)*
  A Profile Specification may reference another Profile Specification. Each referenced Profile Specification must be identified by a `profile-spec-ref` element. A `profile-spec-ref` element contains the following sub-elements.
  - A `description` element.
    This is an optional informational element.
  - A `profile-spec-name` element, a `profile-spec-vendor` element, and a `profile-spec-version` element.
    These elements uniquely identify the Profile Specification, and must match exactly the corresponding `profile-spec-name`, `profile-spec-vendor`, and `profile-version` elements specified in the referenced Profile Specification's `profile-spec` element.
- Zero or more `collator` elements. *(Added in 1.1)*
  These elements are optional. Each collator element specifies a collator that can be used for locale-sensitive comparisons of Profile attributes (refer to the J2SE `java.text.Collator` class). It contains the following attributes and sub-elements:
  - A `strength` attribute.
    This optional attribute specifies the strength property of the collator.

- A `decomposition` attribute.
  This optional attribute specifies the decomposition mode of the collator.
- A `description` element.
  This is an optional informational element.
- A `collator-alias` element.
  It assigns an alias to the collator. Other elements in the same `profile-spec` element use this alias to reference the collator. It allows other elements in the enclosing `profile-spec` element to refer to the collator without specifying the collator's locale and strength and decomposition properties.
- A `locale-language` element.
  It specifies an ISO language code that identifies the language of the locale for the collator. These codes are the lower-case, two-letter codes as defined by ISO-639.
- A `locale-country` element.
  This element is optional. It specifies an ISO country code that identifies a specific country for the locale language. These codes are the upper-case, two-letter codes as defined by ISO-3166.
- A `locale-variant` element.
  This element is optional. It specifies a vendor or browser-specific variant to a locale-language. If this element is specified the locale-country element must also be specified.

- A `profile-classes` element.
  This element contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - A `profile-cmp-interface` element.
    It contains the following elements:
    - A `description` element.
      This is an optional informational element.
    - A `profile-cmp-interface-name` element.
      This element identifies the class name of the Profile CMP interface.
    - Zero or more `cmp-field` elements.
      These elements are option. They must be specified for any Profile CMP fields that have characteristics differing from the deployment descriptor defaults. Each cmp-field element contains the following:
      - A `description` element.
        This is an optional informational element.
      - A `cmp-field-name` element.
        This element identifies the Profile CMP field. For more information see section 10.6.
      - A `unique` attribute.
        This boolean attribute specifies whether a particular value of this attribute can only appear at most once in the Profile Table.
        If not specified, the default value is "False".
        If the Java type of the attribute is an array and the `unique` attribute is "True", a particular value stored in an array element can also only appear once in the Profile Table, i.e. no two array elements of the same attribute stored in the same Profile or different Profiles within the same Profile Table can have the same value.
        The default profile is not considered when evaluating uniqueness.

The `null` value is exempt from these uniqueness constraints and may appear multiple times as a value of this attribute in multiple Profiles within the Profile Table.
This attribute may only be specified for CMP fields whose Java type is a primitive type, an Object wrapper of a primitive type (e.g. `java.lang.Integer`), `java.lang.String`, `javax.slee.Address`, or arrays of these types.

- A `unique-collator-ref` attribute.
  This optional attribute applies only when the `unique` attribute is "True", and the Java type of the Profile CMP field is `java.lang.String`. It references a collator by its `collator-alias` that is specified within the same `profile-spec` element. It is used to determine equality of the CMP field between the various Profiles within the Profile Table. If this attribute is not specified, and the Java type of the CMP field is `java.lang.String` then the `String.equals()` method is used for determining equality.
- Zero or more `index-hint` elements.
  These elements are optional. They are used to provide information to the SLEE about the types of indexes that should be maintained for this Profile CMP field. This allows the SLEE to rapidly locate profiles during query operations.
  These elements may only be specified for CMP fields whose Java type is a primitive type, an Object wrapper of a primitive type (e.g. `java.lang.Integer`), `java.lang.String`, or `javax.slee.Address`.
  Each `index-hint` element contains the following attributes:
  - A `query-operator` attribute.
    This attribute specifies a query operator for the CMP field. Valid query operators are one of the following: "`equals`", "`not-equals`", "`less-than`", "`less-than-or-equals`", "`greater-than`", "`greater-than-or-equals`", "`range-match`", "`longest-prefix-match`", and "`has-prefix`".
  - A `collator-ref` attribute.
    This attribute is optional. It applies only when the Java type of the Profile CMP field is `java.lang.String`. This attribute references a collator by its `collator-alias` that is specified within the same `profile-spec` element. It is used to determine *equality and ordering* of the CMP field between the various Profiles within the Profile Table. If this attribute is not specified, and the Java type of the CMP field is `java.lang.String` then the `String.equals()` method is used for determining equality, and `String.compareTo(String)` is used to determine ordering.

- A `profile-local-interface` element *(Added in 1.1)*
  This element is optional. If the Profile Specification defines a Profile Local interface, this element identifies the Profile Local interface. It contains the following attributes and subelements:

- An `isolate-security-permissions` attribute.
  This attribute controls whether or not security permissions of other protection domains in the call stack are propagated to the Profile when a business method on the Profile Local interface is invoked. If the value of this attribute is "False", then the method in the Profile abstract class invoked as a result of a business method invoked on the Profile Local interface runs with an access control context that includes the protection domain(s) of the Profile Specification as well as the protection domains of any other classes in the call stack as prescribed by the Java security model, such as the SLEE Component that invoked the Profile Local interface method. If the value of the `isolate-security-permissions` attribute is "True", the SLEE automatically wraps the method invoked on the Profile abstract class in response to the Profile Local interface method invocation in an `AccessController.doPrivileged` block in order to isolate the security permissions of the invoked Profile, i.e. the security permissions of other protection domains in the call stack do not affect the invoked Profile.
- A `description` element.
  This is an optional informational element.
- A `profile-local-interface-name` element.
  This element identifies the class name of the Profile Local interface.

o A `profile-management-interface` element.
This element is optional. If the Profile Specification defines a Profile Management interface, this element identifies the Profile Management interface. It contains the following elements:

- A `description` element.
  This is an optional informational element.
- A `profile-management-interface-name` element.
  This element identifies the class name of the Profile Management interface.

o A `profile-abstract-class` element. *(Added in 1.1)*
This element is optional. If the Profile Specification defines a Profile Management abstract class, this element identifies the Profile Management abstract class. It contains the following attributes and sub-elements:

- A `reentrant` attribute.
  This boolean attribute indicates whether the Profiles are re-entrant (see Section 10.11.1), If this attribute is not specified, the default value is "False".
- A `description` element.
  This is an optional informational element.
- A `profile-abstract-class-name` element.
  This element identifies the class name of the Profile abstract class.

o A `profile-table-interface` element. *(Added in 1.1)*
This element is optional. If the Profile Specification defines a Profile Table interface, this element identifies the Profile Table interface. It contains the following elements:

- A `description` element.
  This is an optional informational element.
- A `profile-table-interface-name` element.
  This element identifies the class name of the Profile Table interface.

o A `profile-usage-parameter-interface` element. *(Added in 1.1)*
This element is optional. If the Profile Specification developer defines a Profile Usage

Parameters interface, this element identifies the Profile Usage Parameters interface. It contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- A `profile-usage-parameters-interface-name` element.
  This element identifies the class name of the Profile Usage Parameters interface.

- Zero or more `env-entry` elements. *(Added in 1.1)*
  These elements are optional. Each `env-entry` element binds an environment entry into the JNDI component environment of Profiles of the Profile Specification (see Section 10.19). Each `env-entry` element contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - An `env-entry–name` element.
    This element specifies the location within the JNDI component environment to which the environment entry value will be bound.
  - An `env-entry–type` element.
    This element specifies the type of the value specified by the environment `env-entry` element.
  - An `env-entry-value` element.
    This element specifies the value that will be bound to the location specified by the `env-entry–name` element.
- Zero or more `query` elements. *(Added in 1.1)*
  Each of these elements describes a static query that can be executed by SLEE Components against Profiles in a Profile Table created from the Profile Specification. It contains the following attributes and sub-elements:
  - The `name` attribute.
    This attribute specifies the name of the static query. It is used to derive the method name of the static query method defined in the Profile Specification's Profile Table interface.
  - Zero or more `query-parameter` elements.
    Each of these elements identify a parameter to the static query. It contains the following attributes:
    - The `name` attribute.
      It specifies the name of the query parameter.
    - The `type` attribute.
      It specifies the Java type of the query parameter. The type must be either a primitive type such as an `int` or `long`, or the fully-qualified name of a class, such as `java.lang.String`.
  - A query term such as `compare`, `range-match`, `longest-prefix-match`, `has-prefix`, `and`, `or`, or `not`.
    The static query expression is described using one or more query terms. See Section 10.20.2 for a complete description of each query term.

3.3.8 Recommended Profile interfaces and classes naming convention

The SLEE specification recommends the following naming convention for the interfaces and classes of a Profile Specification.

Class or interface	*Recommended Naming Convention*
Profile CMP interface	<*name*>`ProfileCMP`

Profile Local interface	<*name*>`ProfileLocal`
Profile Management interface	<*name*>`ProfileManagement`
Profile abstract class	<*name*>`ProfileImpl`
Profile Table interface	<*name*>`ProfileTable`

Changed in 1.1: Recommended naming conventions revised for the 1.1 Profile contract.

3.3.9 Profile Specification jar file

The Profile Specification jar file is the standard format for packaging one or more Profile Specifications. It must include the following:

- A Profile Specification jar deployment descriptor.
  - The Profile Specification jar deployment descriptor is stored with the name `META-INF/profile-spec-jar.xml` in the Profile Specification jar file.
  - The root element of the Profile Specification jar deployment descriptor is a `profile-spec-jar` element. This element has the following sub-elements.
    - A `description` element.
      This is an optional informational element.
    - One or more `profile-spec` elements.
      Each of these elements is a Profile Specification's deployment descriptor as defined in Section 3.3.5.
    - A `security-permissions` element.
      This element is optional and contains sub-elements that identify additional security permissions that should be granted to the classes in the Profile Specification jar file (only).
      - A `description` element.
        This is an optional informational element.
      - A `security-permission-spec` element.
        This element identifies the security permission policies used by the Profile Specification jar file classes. The syntax of this element is defined at the following URL:
        `http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html#FileSyntax`.
        Note: If the `codeBase` argument is not specified for a `grant` entry in the provided security permission specification, the SLEE assumes the code base to be the Profile Specification jar file, and the security permissions specified in the `grant` entry are granted to all classes loaded from the Profile Specification jar file, i.e. they apply to all Profile Specifications defined in the Profile Specification jar file. The security permissions are not granted to classes loaded from any other dependent component jar required by the Profile Specifications defined in the Profile Specification deployment descriptor. If the `codeBase` argument is specified for a `grant` entry, the specified path is taken to be relative to the root directory of the Profile Specification jar within the deployable unit jar, but its use is otherwise undefined by the SLEE specification.
- Class files of the Profile Specifications specified by the `profile-spec` elements of the `profile-spec-jar` element.
  - The Profile Specification jar file must contain, either by inclusion or by reference, the class files of each Profile Specification.

- The Profile Specification jar file must also contain, either by inclusion or by reference, the class files for all the classes and interfaces that the Profile Specification's classes and interfaces depend on, except J2SE classes and SLEE classes. This includes their super-classes and superinterfaces, dependent classes, and the classes and interfaces used as method parameters, results, and exceptions.

3.3.10 Profile Specification jar file example

The following example illustrates a Profile Specification jar file that contains a BarAddressProfileSpec. This example also shows the naming convention described in Section 3.3.8 and the deployment descriptor of the BarAddressProfileSpec.

The deployment descriptor for the example Profile Specification jar file is as follows:

```
<profile-spec-jar>
       <profile-spec>
              …
              <profile-spec-name> BarAddressProfileSpec </profile-spec-name>
              <profile-spec-vendor> com.foobar </profile-spec-vendor>
              <profile-spec-version> 3.1.1 </profile-spec-version>
              <profile-classes>
                     <profile-cmp-interface>
                            <description> ... </description>
                            <profile-cmp-interface-name>
                                   com.foobar.BarAddressProfileCMP
                            </profile-cmp-interface-name>
                     </profile-cmp-interface>
                     <profile-local-interface>
                            <description> ... </description>
                            <profile-local-interface-name>
                                   com.foobar.BarAddressProfileLocal
                            </profile-local-interface-name>
                     </profile-local-interface>
                     <profile-management-interface>
                            <description> ... </description>
                            <profile-management-interface-name>
                                   com.foobar.BarAddressProfileManagement
                            </profile-management-interface-name>
                     </profile-management-interface>
                     <profile-abstract-class>
                            <description> ... </description>
                            <profile-abstract-class-name>
                                   com.foobar.BarAddressProfileImpl
                            </profile-abstract-class-name>
                     </profile-abstract-class>
              </profile-classes>
              …
       </profile-spec>
</profile-spec-jar>
```

The content of the Profile Specification jar file is as follows:

```
META-INF/profile-spec-jar.xml
META-INF/MANIFEST.MF
…
com/foobar/BarAddressProfileCMP.class
com/foobar/BarAddressProfileLocal.class
com/foobar/BarAddressProfileManagement.class
com/foobar/BarAddressProfileImpl.class
…
```

3.4 Create a Service

Changed in 1.1: The SBB details described below follow the SLEE 1.1 contract for Services. For a similar description using the SLEE 1.0 contract for Services, the reader is referred to the SLEE 1.0 specification. A 1.1 compliant SLEE must also implement the 1.0 Service contract. A 1.1 SLEE must use the DTD referenced by a Service's deployment descriptor to determine whether the SBB follows the 1.0 or 1.1 contract.

Use of 1.0 Services should be avoided as this backward compatibility requirement may be removed in a future version of the SLEE specification

The Service Deployer creates a Service by:

- Defining the Service.
- Customizing the SBBs of the Service to the target operational environment.
- Packaging the Service in a deployable unit.

3.4.1 Service deployment descriptor

The Service Deployer specifies the following aspects of the Service in the Service deployment descriptor. A Service deployment descriptor contains a `service` element. The `service` element contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- A `service-name` element, a `service-vendor` element, and a `service-version` element.
  These elements identify the Service.
- A `root-sbb` element.
  This element identifies the root SBB of the Service. A `root-sbb` element contains the following sub-elements.
  - A `description` element.
    This is an optional informational element.
  - An `sbb-name` element, an `sbb-vendor` element, and an `sbb-version` element.
    These elements uniquely identify the root SBB.
- A `default-priority` element.
  This element specifies the default event delivery priority for the child relation from the SLEE to root SBB.
- An `address-profile-table` element.
  This element is optional. This Profile Table contains provisioned addresses that may cause new root SBB entities of the Service to be instantiated. This element specifies the Address Profile Table of the Service. The Profile Specification of this Profile Table must be the same as the Profile Specification specified by the `address-profile-spec-alias-ref` of the root SBB.
- A `resource-info-profile-table` element.
  Deprecated in 1.1: The Resource Info Profile Table has been deprecated The Resource Adaptor architecture defined herein describes how resource adaptors may interact with Profile Tables and Profiles. SBB and/or Service developers should avoid the use of the Resource Info Profile Table as it may be removed in a future version of this specification.
  This element is optional. This table contains provisioned data that should be passed to resource adaptor entities when the Service is in running. This element specifies the Resource Info Profile Table of the Service. The Profile Specification of this Profile Table must be the SLEE specification defined Resource Info Profile Specification.

3.4.2 Component identity of a Service

The component identity of Service uniquely identifies a Service. A Service's name, vendor, and version (as defined by the `service-name`, `service-vendor`, and `service-version` deployment descriptor elements) determine the component identity of a Service.

The `ServiceID` class (see Section 14.6.5.5) defines the interface of a Java object that encapsulates a Service's component identity. This Java object is also known as a Service component identifier. *(Changed in 1.1: `ServiceID` interface replaced by `ServiceID` class)*

3.4.3 Customizing the Service to the operational environment

The Service Deployer can also customize the root SBBs and their child SBBs to the target operational environment. The Service Deployer should have a good understanding of the target operational environment. The target operational environment includes the available resource adaptor types, resource adaptors, resource adaptor entities, other SBBs and Services that have already been installed.

The Service Deployer can adapt and customize an SBB to the target operational environment by modifying deployment descriptor elements of the SBB. The Service Deployer may have to disassemble SBB jar files to modify these deployment descriptor elements or the SLEE may provide tools to modify these deployment descriptor elements.

The Service Deployer can add and modify the following SBB deployment descriptor elements:

- `activity-context-attribute-alias` elements.
  Two independent sets of SBBs developed independently may choose the same alias name for two different logical attributes. The Service Deployer can resolve Activity Context attribute alias name conflicts by modifying `attribute-alias-name` elements.
- `env-entry` elements.
- `resource-adaptor-type-binding` elements.
- `resource-adaptor-entity-binding` elements.
- `ejb-ref` elements.

The Service Deployer should avoid modifying other SBB deployment descriptor elements.

3.4.4 Packaging the Service for deployment

The SBB Developer distributes one or more Service deployment descriptors in a Service XML deployment descriptor file. This file contains a single `service-xml` element. The `service-xml` element contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- One or more `service` elements.
  Each of these elements is a Service's deployment descriptor as defined in Section 3.4.1.

An example Service XML deployment descriptor file may be as follows:

```
<service-xml>
        …
        <service>
                <description>
                        …
                </description>
                <service-name> FooService </service-name>
                <service-vendor> com.foobar </service-vendor>
                <service-version> 1.0 </service-version>
                <root-sbb>
                        <description> … </description>
                        <sbb-name> Foo SBB </sbb-name>
                        <sbb-vendor> com.foobar </sbb-vendor>
                        <sbb-version> 1.0 </sbb-version>
                </root-sbb>
                <default-priority> … </default-priority>
                <address-profile-table> … </address-profile-table>
                …
        </service>
```

```
        …
</service-xml>
```

3.4.5 Deploying components

Before event types, Profile Specifications, SBBs and Services can be deployed or installed into a SLEE, the Service Deployer must create a deployable unit that contains the components that are to be deployed.

A deployable unit is a jar file. This jar file contains a deployable unit deployment descriptor, the constituent jar files and Service XML files of the deployable unit. The deployable unit deployment descriptor is stored with the name `META-INF/deployable-unit.xml` in the deployable unit jar file.

A deployable unit deployment descriptor contains a `deployable-unit` element. The `deployable-unit` element contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- One or more `jar` elements.
  Each `jar` element contains the name of a constituent jar file included in the deployable unit jar file. This name is relative to the root of the deployable unit jar file. The constituent jar file may be an SBB jar file, Profile Specification jar file, an event jar file, a resource adaptor type jar file, a resource adaptor jar file, or a library jar file. The type of the constituent jar file is determined from the constituent jar file.
- Zero or more `service-xml` deployment descriptor elements.
  Each of these elements specifies the name of a Service XML deployment descriptor file. This name is relative to the root of the deployable unit jar file.

Event types and Profile Specifications that may potentially be referenced by multiple deployable units should be packaged in independent deployable units. If these components are packaged in a deployable unit that contains a Service, for example, then the Service cannot be uninstalled independently without uninstalling those components that other deployable units may depend on.

3.4.6 Deployable unit jar file example

The following example illustrates a deployable unit jar file.

The deployment descriptor for the example deployable unit jar file is as follows:

```
<deployable-unit>
        <description> … </description>
        …
        <jar> SomeProfileSpec.jar </jar>
        <jar> BarAddressProfileSpec.jar </jar>
        <jar> SomeCustomEvent.jar </jar>
        <jar> FooSBB.jar </jar>
        <jar> BarSBB.jar </jar>
        …
        <service-xml> FooService.xml </service-xml>
        …
</deployable-unit>
```

The content of the deployable unit jar file is as follows:

```
META-INF/deployable-unit.xml
META-INF/MANIFEST.MF
…
SomeProfileSpec.jar
BarAddressProfileSpec.jar
SomeCustomEvent.jar
FooSBB.jar
BarSBB.jar
FooService.xml
…
```

3.5 Administer Services and Profiles

The Administrator can perform the following management operations through the management interfaces of the SLEE:

- Operations on deployable units.
  - Install and uninstall a deployable unit.
  - Find out what components are in an installed deployable unit.
- Operations on Services.
  - Activate and deactivate a Service.
  - Get the life cycle state of a Service.
  - List the Services in a particular life cycle state.
  - Get and reset the usage parameters of SBBs in a Service.
  - Get and set the trace filter level for tracers used by SBBs in a Service. (Added in 1.1)
- Operations on Profiles.
  - Create Profile Tables from Profile Specifications.
  - Add and remove Profiles from Profile Tables.
  - Read and modify Profiles.
  - Get and set the trace filter levels for tracers used by Profiles in a Profile Table. *(Added in 1.1)*

Typically, the Administrator performs the following sequence of management operations on a Service:

- Install the deployable unit jar file provided by the Service Deployer into the SLEE.
  This deployable unit includes the Service deployment descriptor and the SBB jar files of the SBBs of the Service.
- Create the Profile Tables required by the Service.
  If the Service specifies an Address Profile Table, the specified Address Profile Table should be created before the Service is activated. If the Service specifies a Resource Info Profile Table, the specified Resource Info Profile Table should be created before the Service is activated. If any SBBs in the Service require any other Profile Tables to function correctly, they should also be created before the Service is activated. *(Clarified in 1.1)*
- Populate Profile Tables with Profiles.
  The Administrator may populate Profile Tables that contain common configuration data for the SBBs of the Service.
- Activate the Service.
  At this point, the SLEE may create new root SBB entities for the Service to process events.
- Add, remove and modify Profiles.
  Profiles typically contain provisioned data that represent subscribers. For example, Address Profiles may be added or removed as subscribers are added or removed. A subscriber's data may be stored in one or more Profiles in one or more Profile Tables. The Administrator adds, removes and modifies these Profiles as subscribers are added, removed, or when their subscriber data changes.
- Deactivate the Service.
  At this point, no new root SBB entities are created for the service.
- Uninstall the deployable unit.
  After the SLEE uninstalls the deployable unit, it loses all knowledge of the Services in the deployable unit.

Chapter 4 Addresses

An address is a unique identifier for an entity in a network or the SLEE. A common address today is a telephone number or a URI address.

4.1 `Address` objects

The SLEE uses an `Address` object to represent an address. An `Address` object is an instance of the `Address` class. The SLEE APIs pass `Address` objects wherever a method returns one or more addresses or requires one or more addresses as input arguments. The `Address` class allows resources to use the same `Address` object for different resources and/or resource adapters.

The public interface of the `Address` class is as follows.

```
package javax.slee;

public class Address {
        public Address(AddressPlan addressPlan,
                       String addressString) { … }
        public Address(AddressPlan addressPlan,
                       String addressString,
                       AddressPresentation addressPresentation,
                       AddressScreening addressScreening,
                       String subAddressString,
                String addressName) { … }
        public AddressPlan getAddressPlan() { … }
        public String getAddressString(){ … }
        public AddressPresentation getAddressPresentation() { … }
        public AddressScreening getAddressScreening() { … }
        public String getSubAddressString() { … }
        public String getAddressName() { … }
        public String toString() { … }
        public int hashCode() { … }
        public boolean equals(Object o) { … }
}
```

- The `addressPlan` attribute.
  It specifies the address plan of the address. The address plan determines the format of the address string and how it should be interpreted. It is mandatory in all addresses.
- The `addressString` attribute.
  It specifies the address string of the address. The address string is the unique address value that identifies an end user in a specific type of network. It is mandatory in all addresses.
- The `addressPresentation` attribute.
  It specifies the address presentation of the address. The address presentation is optional and the value of this attribute is `null` when an address presentation is not provided.
- The `addressScreening` attribute.
  It specifies the address screening of the address. The address screening is optional and the value of this attribute is `null` when an address screening is not provided.
- The `subAddressString` attribute.
  It specifies the sub-address of the address. Some protocols allow sub-addressing (e.g. extensions) within an address. The sub-address is optional and the value of this attribute is `null` when a sub-address is not provided.
- The `addressName` attribute.
  It specifies the address name of the address. Some systems allow a personal name to be displayed instead of the actual address string. The address name is optional and the value of this attribute is `null` when an address name is not provided.

The following rules apply to `Address` objects.

- An `Address` object is immutable. Once it has been created, its contents cannot be changed.
- An `Address` object must at least have both a non-`null` address plan and a non-`null` address string. The address string should be of the format defined by the address plan[5]. Either public constructor can be used to initialize an `Address` object with a valid address plan and address string.

4.2 `AddressPlan` objects

Changed in 1.1: String representations and a fromString method have been added.

The `AddressPlan` class defines an enumerated type for address plans.

The address plan of an address determines the format and structure of the address string specified in the `addressString` attribute of the `Address` object. The SLEE supports the JAIN SPA Common specification address plans and the following additional address plans, H323, Globaltitle, Subsystem Number (SSN), E164_Mobile, SLEE Profile Tables and SLEE Profiles.

The public interface of the `AddressPlan` class is as follows.

```
package javax.slee;

public class AddressPlan {
        // singletons
        public static AddressPlan NOT_PRESENT …;
        public static AddressPlan UNDEFINED …;
        public static AddressPlan IP …;
        public static AddressPlan MULTICAST …;
        public static AddressPlan UNICAST …;
        public static AddressPlan E164 …;
        public static AddressPlan AESA …;
        public static AddressPlan URI …;
        public static AddressPlan NSAP …;
        public static AddressPlan SMTP …;
        public static AddressPlan E164_MOBILE …;
        public static AddressPlan X400 …;
        public static AddressPlan SIP …;
        public static AddressPlan H323 …;
        public static AddressPlan GT …;
        public static AddressPlan SSN …;
        public static AddressPlan SLEE_PROFILE_TABLE …;
        public static AddressPlan SLEE_PROFILE …;

        // integer representation
        public static int ADDRESS_PLAN_NOT_PRESENT …;
        public static int ADDRESS_PLAN_UNDEFINED …;
        public static int ADDRESS_PLAN_IP …;
        public static int ADDRESS_PLAN_MULTICAST …;
        public static int ADDRESS_PLAN_UNICAST …;
        public static int ADDRESS_PLAN_E164 …;
        public static int ADDRESS_PLAN_AESA …;
        public static int ADDRESS_PLAN_URI …;
        public static int ADDRESS_PLAN_NSAP …;
        public static int ADDRESS_PLAN_SMTP …;
        public static int ADDRESS_PLAN_E164_MOBILE …;
        public static int ADDRESS_PLAN_X400 …;
        public static int ADDRESS_PLAN_SIP …;
        public static int ADDRESS_PLAN_H323 …;
        public static int ADDRESS_PLAN_GT …;
        public static int ADDRESS_PLAN_SSN …;
        public static int ADDRESS_PLAN_SLEE_PROFILE_TABLE …;
        public static int ADDRESS_PLAN_SLEE_PROFILE …;
```

[5] The `javax.slee.Address` class provided by the SLEE specification performs no verification on the address string passes to its constructor to check that it correctly follows the syntax of the specified address plan.

```
        // string representation
        public static String NOT_PRESENT_STRING = “Not Present”;
        public static String UNDEFINED_STRING = “Undefined”;
        public static String IP_STRING = “IP”;
        public static String MULTICAST_STRING = “Multicast”;
        public static String UNICAST_STRING = “Unicast”;
        public static String E164_STRING = “E.164”;
        public static String AESA_STRING = “AESA”;
        public static String URI_STRING = “URI”;
        public static String NSAP_STRING = “NSAP”;
        public static String SMTP_STRING = “SMTP”;
        public static String E164_MOBILE_STRING = “E.164 Mobile”;
        public static String X400_STRING = “X400”;
        public static String SIP_STRING = “SIP”;
        public static String H323_STRING = “H323”;
        public static String GT_STRING = “GT”;
        public static String SSN_STRING = “SSN”;
        public static String SLEE_PROFILE_TABLE_STRING = “SLEE Profile Table”;
        public static String SLEE_PROFILE_STRING = “SLEE Profile”;


        public boolean isNotPresent() { … }
        public boolean isUndefined() { … }
        public boolean isIP() { … }
        public boolean isMulticast() { … }
        public boolean isUnicast() { … }
        public boolean isE164() { … }
        public boolean isAESA() { … }
        public boolean isURI() { … }
        public boolean isNSAP() { … }
        public boolean isSMTP() { … }
        public boolean isE164Mobile() { … }
        public boolean isX400() { … }
        public boolean isSIP() { … }
        public boolean isH323() { … }
        public boolean isGT() { … }
        public boolean isSSN() { … }
        public boolean isSleeProfileTable() { … }
        public boolean isSleeProfile() { … }

        public static AddressPlan fromString(String plan)
               throws NullPointerException, IllegalArgumentException { … }
        public static AddressPlan fromInt(int plan)
               throws IllegalArgumentException { … }
        public int toInt() { … }
        public int hashCode() { … }
        public boolean equals(Object o) { … }
        public String toString() { … }
}
```

- A singleton instance of each enumerated value is guaranteed (via an implementation of the `readResolve` method, see `java.io.Serializable` for more details), so that equality tests using == are always evaluated correctly. For convenience, an integer representation of `AddressPlan` objects is also available.
- Each of the `is<`*`Plan`*`>` methods determines if this `AddressPlan` object represents the *<PLAN>* of the address, and is equivalent to `(this == <`*`PLAN`*`>)`. For example, the `isSIP` method determines if this `AddressPlan` object represents the SIP plan of the address and is equivalent to `(this == SIP)`.
- The `fromInt` and `toInt` methods allow conversion between the `AddressPlan` object form and numeric form.
- The `fromInt` method throws a `java.lang.IllegalArgumentException` if the `plan` argument is not one of the integer address plan representations.

- The `fromString` method allows conversion from a `String` to an `AddressPlan`. The conversion is case insensitive. This method throws a `java.lang.NullPointerException` if the argument is `null`. This method throws a `java.lang.IllegalArgumentException` if the argument does not specify an address plan that is known by this class.

4.3 Address string

The Java type of the `addressString` attribute is `java.lang.String`. It contains an address string. An address string is a string of characters or decimal digits segmented into groups in order to identify specific elements used for identification, routing and charging capabilities. Each address plan defines the format of address strings that are valid for the address plan. This attribute cannot contain wildcards. Examples of valid address strings for three address plans follows:

- An address with an URI address plan specifies a protocol, a domain and an optional path information and has the format "protocol://domain[/path]", e.g. "http://www.foobar.com/index.html".
- An address with an address plan of type SIP has an address string in the format of "sip:[name][@domain]", e.g. "sip:slee@foobar.com".
- The address string of an address with an address plan of type E164 should be an international telephone number without the international access code, including the country code and excluding the leading zero of the area code, e.g., "1650999999" for a US based number.

4.4 `AddressPresentation` objects

Changed in 1.1: String representations and a fromString method have been added.

The `AddressPresentation` class defines an enumerated type for address presentations. The address presentation of an address defines whether an address can be presented to other call parties. The user application determines whether the address is to be presented to other end-users.

The public interface of the `AddressPresentation` class is as follows.

```
package javax.slee;

public class AddressPresentation {
        // singletons
        public static AddressPresentation ALLOWED …;
        public static AddressPresentation RESTRICTED …;
        public static AddressPresentation UNDEFINED …;
        public static AddressPresentation ADDRESS_NOT_AVAILABLE …;

        // integer representation
        public static int ADDRESS_PRESENTATION_ALLOWED …;
        public static int ADDRESS_PRESENTATION_RESTRICTED …;
        public static int ADDRESS_PRESENTATION_UNDEFINED …;
        public static int ADDRESS_PRESENTATION_ADDRESS_NOT_AVAILABLE …;

        // string representation
        public static String ALLOWED_STRING = "Allowed";
        public static String RESTRICTED_STRING = "Restricted";
        public static String UNDEFINED_STRING = "Undefined";
        public static String ADDRESS_NOT_AVAILABLE_STRING = "Address Not Available";


        public boolean isAddressNotAvailable() { … }
        public boolean isUndefined() { … }
        public boolean isRestricted() { … }
        public boolean isAllowed() { … }

        public static AddressPresentation fromString(String presentation)
                throws NullPointerException, IllegalArgumentException { … }
        public static AddressPresentation fromInt(int presentation)
                throws IllegalArgumentException { … }
        public int toInt() { … } …
```

```
        public int hashCode() { … }
        public boolean equals(Object o) { … }
        public String toString() { … }
}
```

- A singleton instance of each enumerated value is guaranteed (via an implementation of the `readResolve` method, see `java.io.Serializable` for more details), so that equality tests using == are always evaluated correctly. For convenience, an integer representation of `AddressPresentation` objects is also available.
- Each of the `is<`*`Presentation`*`>` methods determines if this `AddressPresentation` object represents the `<`*`PRESENTATION`*`>` of the address, and is equivalent to `(this == <`*`PRESENTATION`*`>)`. For example, the `isAllowed` method determines if this `AddressPresentation` object represents the Allowed presentation of the address and is equivalent to `(this == ALLOWED)`.
- The `fromInt` and `toInt` methods allow conversion between the `AddressPresentation` object form and numeric form.
- The `fromInt` method throws a `java.lang.IllegalArgumentException` if the `presentation` argument is not one of the integer address presentation representations.
- The `fromString` method allows conversion from a `String` to an `AddressPresentation`. The conversion is case insensitive. This method throws a `java.lang.NullPointerException` if the argument is `null`. This method throws a `java.lang.IllegalArgumentException` if the argument does not specify an address presentation that is known by this class.

4.5 `AddressScreening` objects

Changed in 1.1: String representations and a fromString method have been added.

The `AddressScreening` class defines an enumerated type for the address screenings. The address screening of an address defines whether an address has been screened by a user application. The user can optionally choose one of these constants depending on whether they have screened their address and the outcome they received. The address can be screened either by the network or a user or the address screening may be undefined.

The public interface of the `AddressScreening` class is as follows.

```
package javax.slee;

public class AddressScreening {

        // singletons
        public static AddressScreening NETWORK …;
        public static AddressScreening UNDEFINED …;
        public static AddressScreening USER_NOT_VERIFIED …;
        public static AddressScreening USER_VERIFIED_PASSED …;
        public static AddressScreening USER_VERIFIED_FAILED …;

        // integer representation
        public static int ADDRESS_SCREENING_NETWORK …;
        public static int ADDRESS_SCREENING_UNDEFINED …;
        public static int ADDRESS_SCREENING_USER_NOT_VERIFIED …;
        public static int ADDRESS_SCREENING_USER_VERIFIED_PASSED …;
        public static int ADDRESS_SCREENING_USER_VERIFIED_FAILED …;

        // string representation
        public static String NETWORK_STRING = “Network”;
        public static String UNDEFINED_STRING = “Undefined”;
        public static String USER_NOT_VERIFIED_STRING = “User Not Verified”;
        public static String USER_VERIFIED_PASSED_STRING = “User Verified Passed”;
        public static String USER_VERIFIED_FAILED_STRING = “User Verified Failed”;
```

```
        public boolean isNetwork() { … }
        public boolean isUndefined() { … }
        public boolean isUserNotVerified() { … }
        public boolean isUserVerifiedPassed() { … }
        public boolean isUserVerifiedFailed() { … }

        public static String fromString(String screening)
               throws NullPointerException, IllegalArgumentException { … }
        public static AddressScreening fromInt(int screening)
               throws IllegalArgumentException { … }
        public int toInt() { … }
        public int hashCode() { … }
        public boolean equals(Object o) { … }
        public String toString() { … }
}
```

- A singleton instance of each enumerated value is guaranteed (via an implementation of the `readResolve` method, see `java.io.Serializable` for more details), so that equality tests using == are always evaluated correctly. For convenience, an integer representation of `AddressScreening` objects is also available.
- Each of the `is<`*`Screening`*`>` methods determines if this `AddressScreening` object represents the <*SCREENING*> of the address, and is equivalent to `(this == <`*`SCREENING`*`>)`. For example, the `isNetwork` method determines if this `AddressScreening` object represents the Network screening of the address and is equivalent to `(this == NETWORK)`.
- The `fromInt` and `toInt` methods allow conversion between the `AddressScreening` object form and numeric form.
- The `fromInt` method throws a `java.lang.IllegalArgumentException` if the `screening` argument is not one of the integer address presentation representations.
- The `fromString` method allows conversion from a `String` to an `AddressPresentation`. The conversion is case insensitive. This method throws a `java.lang.NullPointerException` if the argument is `null`. This method throws a `java.lang.IllegalArgumentException` if the argument does not specify an address screening that is known by this class.

4.6 Address example

An example email address that can be presented to the user and has been screened by the network has the following attribute values:

- `addressPlan = AddressPlan.SMTP`
- `addressString = ”alice@jcp.org”`
- `addressScreening = AddressScreening.NETWORK`
- `addressPresentation = AddressPresentation.ALLOWED`
- `addressName = ”Alice”`

Chapter 5 SBB Local Interface

An SBB entity can invoke a target SBB entity in a synchronous manner through the SBB local interface of the target SBB. This interface is known as the SBB local interface because the SBB object that represents the caller and the SBB object that represents the callee must be collocated in the same JVM. In order to be invoked synchronously, an SBB must declare an SBB local interface. The SBB local interface declares the methods of the SBB that may be invoked synchronously. An SBB local object is an instance of a SLEE implemented class that implements an SBB local interface.

5.1 How to obtain an SBB local object

An SBB can obtain an SBB local object that represents an SBB entity via the following mechanisms:

- Receive the SBB local object as the result of invoking the `create` method on a `ChildRelation` object.
- Receive the SBB local object as the result of invoking the `getSbbLocalObject` method on an `SbbContext` object.
- Receive the SBB local object as a result of a method call on an SBB local object. The callee may receive the SBB local object through one of its input arguments. The caller may receive the SBB local object through the return argument.
- Retrieve an SBB local object from a CMP field (see Section 6.5).
- Receive a `ChildRelation` object that implements the `java.util.Collection` interface. This `ChildRelation` object and its `Iterator` objects provide access to SBB local objects.

5.2 What can be done with an SBB local object

An SBB object holding a reference to an SBB local object may do any of the following:

- Test if the SBB local object represents the same SBB entity as another SBB local object.
- Remove the SBB entity (and the SBB entity's descendents) represented by the SBB local object.
- Set and get the event delivery priority of the SBB entity represented by the SBB local object.
- Invoke an SBB Developer defined method of the SBB local object.
- Pass the SBB local object in an input or return argument of a local method call.
- Store the SBB local object into a CMP field.

5.3 A typical scenario

In a typical scenario, an SBB object that represents a parent SBB entity creates a child SBB entity by invoking the `create` method on one of its `ChildRelation` objects. The `create` method returns an object that implements the SBB local interface of the child SBB entity. This returned object is an SBB local object that represents the newly created child SBB entity. Zero or more SBB local objects may represent the same SBB entity.

When the parent or another SBB invokes an SBB Developer defined method on the SBB local object, the SBB local object delegates the method invocation to a local SBB object that represents the target SBB entity. The SLEE is responsible for providing a suitable local SBB object to receive this method invocation. If a suitable local SBB object does not exist, the SLEE may have to create a new SBB object or obtain an existing SBB object in the Pooled state, and assign the SBB object to the target SBB entity.

When the SBB entity represented by the SBB local object no longer exists (because it has been deleted), a method invocation on the SBB local object will:

- Mark the current transaction for rollback, and

- Return by throwing the `javax.slee.TransactionRolledbackLocalException`.

5.4 Pass-by-reference semantics

The arguments of the methods of the SBB local interface are passed by reference. An SBB that defines an SBB local interface must be coded to assume that the caller and the callee may potentially reference the same Java object that is passed directly or indirectly as an argument or result. In designing an SBB that makes use of the local programming model offered by SBB local interfaces, the SBB Developer must be aware of the potential multiple references to the same objects passed through the SBB local interface. In particular, the SBB Developer must be careful that one SBB object does not reference Java objects that are intended to be the private state of another SBB object. The caller and callee SBB object may be programmed to rely on pass-by-reference semantics. For example, a caller may have a large document that it wants to pass to the callee to modify (and the callee further passes on). In the local programming model, the sharing of state is possible. On the other hand, when the callee wants to return a data structure to the caller but the caller does not want the callee to modify it, the caller explicitly copies the data structure before passing it.

5.5 The `SbbLocalObject` interface

This interface is the base interface of all SBB local interfaces. All SBB local interfaces extend the `SbbLocalObject` interface. If an SBB does not define an SBB specific local interface, then the SBB's local interface is `SbbLocalObject`.

The SLEE provides the implementation of the methods defined in the `SbbLocalObject` interface.

The `SbbLocalObject` interface is as follows:

```
package javax.slee;

public interface SbbLocalObject {
        public boolean isIdentical(SbbLocalObject obj)
                throws  TransactionRequiredLocalException,
                        SLEEException;
        public void setSbbPriority(byte priority)
                throws  TransactionRolledbackLocalException,
                        TransactionRequiredLocalException,
                        SLEEException;
        public byte getSbbPriority()
                throws  TransactionRolledbackLocalException,
                        TransactionRequiredLocalException,
                        SLEEException;
        public void remove()
                throws  TransactionRolledbackLocalException,
                        TransactionRequiredLocalException,
                        SLEEException;
}
```

5.5.1 `SbbLocalObject` interface `isIdentical` method

This method determines if two SBB local objects represent the same SBB entity.

- The SLEE specification does not specify "object equality" (i.e. use of the `==` operator) for SBB local object references. The result of comparing two SBB local object references using the Java programming language `Object.equals` method is unspecified. Performing the `Object.hashCode` method on two SBB local objects that represent a single SBB entity is not guaranteed to yield the same result. Therefore, the `isIdentical` method should be used to determine if two SBB local objects represent the same SBB entity.
- This method is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.

- This method throws a `javax.slee.SLEEException` if the method failed due to a SLEE level or system level failure.

5.5.2 `SbbLocalObject` interface `setSbbPriority` method

An SBB object invokes the `setSbbPriority` method on an SBB local object that represents a second SBB entity to set the event delivery priority of the second SBB entity relative to the second SBB's siblings (see Section 8.6.7).

- The `priority` argument specifies the new priority. The priority has the range of –128 to 127 inclusive.
- All SBB local objects that represent an SBB entity that does not exist are invalid. An attempted method invocation on an invalid SBB local object marks the current transaction for rollback and throws a `javax.slee.TransactionRolledbackLocalException` (a subclass of `javax.slee.SLEEException`).
- This method is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- This method may also throw the `javax.slee.SLEEException` if the method failed due to a SLEE level or system level failure. If the caller of this method receives this exception, the caller does not know, in general, whether the priority was set or not. The caller also does not know if the transaction has been marked for rollback. However, the caller may determine the priority by using `getSbbPriority` and determine the transaction status by using the `getRollbackOnly` method (see Section 6.10.3).

5.5.3 `SbbLocalObject` interface `getSbbPriority` method

An SBB object invokes the `getSbbPriority` method on an SBB local object that represents the a second SBB entity to get the event delivery priority of the second SBB entity relative to the second SBB entity's siblings (see Section 8.6.7).

- The return value is the second SBB entity's priority. The priority has the range of –128 to 127 inclusive.
- All SBB local objects that represent an SBB entity that does not exist are invalid. An attempted method invocation on an invalid SBB local object marks the current transaction for rollback and throws a `javax.slee.TransactionRolledbackLocalException` (a subclass of `javax.slee.SLEEException`).
- This method is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- This method may also throw the `javax.slee.SLEEException` if the method failed due to a SLEE level or system level failure.

5.5.4 `SbbLocalObject` interface `remove` method

An SBB object invokes the `remove` method on an SBB local object that represents the second SBB entity to initiate a cascading removal of the second SBB entity and the second SBB entity's descendents (see Section 2.2.6).

- When the SLEE performs a cascading removal of an SBB entity sub-tree, it performs a pre-order traversal of the sub-tree to delete the SBB entities in the sub-tree. It deletes the parent SBB entity before it deletes the child SBB entities. The SLEE does not specify the order in which sibling SBB entities are deleted. When the SLEE deletes an SBB entity, it does the following:

  - It detaches the SBB entity from all Activity Contexts.
  - It invokes the appropriate life cycle methods (see Section 6.3) of an SBB object that caches the SBB entity's state.
  - It removes the SBB entity from the `ChildRelation` object that the SBB entity belongs to.
  - It removes the persistent representation of the SBB entity.

  The SLEE performs the above operations of a cascading removal in the same transaction that the `remove` method was invoked in. If the transaction commits, the result of the above operations become permanent. Otherwise, if the transaction does not commit, the changes are discarded.

- All SBB local objects that represent an SBB entity that does not exist are invalid. An attempted method invocation on an invalid SBB local object marks the current transaction for rollback and throws a `javax.slee.TransactionRolledbackLocalException` (a subclass of `javax.slee.SLEEException`).
- This method is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- This method may also throw a `javax.slee.SLEEException` if the method failed due to a SLEE level or system level failure. If the caller of this method receives this exception, the caller does not know, in general, whether the entity object was removed or not. The caller also does not know if the transaction has been marked for rollback. However, the caller may determine the transaction status by using the `getRollbackOnly` method (see Section 6.10.3).

Logically, the SLEE also invokes this method to remove an SBB entity tree. The SLEE invokes this method on the root SBB entity of the SBB entity tree after the attachment count of the root SBB entity decrements to zero. This method initiates a cascading removal of the SBB entity tree. If this method returns by throwing a `RuntimeException`, the SLEE should log this condition. Furthermore, it may forcefully delete the persistent state of the SBB entities in the SBB entity tree, or provide some other mechanism for the Administrator to manually delete the persistent state of these SBB entities. In either case, the SLEE should behave as if the SBB entity tree is successfully removed, i.e. the root SBB entity of the SBB entity tree is no longer a child SBB entity of the SLEE and it will no longer have any effect on event routing.

5.6 The SBB local interface

An SBB may define SBB specific local interface methods in an SBB specific local interface.

- The SBB local interface must be defined in a named package, i.e. the class must have a `package` declaration. *(Added in 1.1)*
- The SBB specific local interface must be `public` and must extend, either directly or indirectly, the `SbbLocalObject` interface.
- All SBBs have an SBB local interface. If the SBB Developer does not provide an SBB local interface for an SBB, then the SBB local interface of the SBB is the generic `SbbLocalObject` interface.
- The names of the SBB specific local interface methods must not begin with "sbb" or "ejb".
- The SLEE provides the implementation of the methods defined in the SBB local interface. More precisely, the SLEE provides a concrete class that implements each SBB local interface. An SBB local object is an instance of this class. The SLEE provided implementations of these methods delegate invocations on an SBB local object that represents an SBB entity to an SBB object that represents the SBB entity (if the SBB entity has not been removed).
- The SBB Developer provides the implementation of SBB Developer defined local interface methods declared in the SBB specific extension of the `SbbLocalObject` interface in the SBB ab-

stract class. For each method defined by the SBB Developer, there must be a matching method in the SBB abstract class. The matching method must have:

- The same name.
- The same number of arguments, and same argument and return types.
- The same set of exceptions in the throws clause.

- All SBB local objects that represent an SBB entity that does not exist are invalid. An attempted invocation on an invalid SBB local object marks the current transaction for rollback and throws a `javax.slee.TransactionRolledbackLocalException` (a subclass of `javax.slee.SLEEException`).
- An SBB Developer defined local interface method is a mandatory transactional method (see Section 9.6.1). The SLEE throws a `javax.slee.TransactionRequiredLocalException` if an SBB Developer defined local interface method of an SBB local object is invoked without a valid transaction context.
- This method may also throw a `javax.slee.SLEEException` if the method failed due to a SLEE level or system level failure. If the caller of this method receives this exception, the caller does not know, in general, whether the corresponding method implementation in the SBB abstract class was invoked. The caller also does not know if the transaction has been marked for rollback. However, the caller may determine the transaction status by using the `getRollbackOnly` method (see Section 6.10.3).
- The SBB Developer defined SBB local interface methods must not throw `java.rmi.RemoteException`, any subclass of `RemoteException`, or any `RuntimeException`. For more information on exception handling for SBB local object invocations, see Section 9.12.1.
- Parameters to local interface methods are passed by reference.
- *Note that the `SbbLocalObject` does not expose the methods of the `javax.slee.Sbb` interface, event handler methods, or the various callback methods. These methods are used by the SLEE to manage SBB object instances, deliver events, and handle callbacks.*

The following example illustrates two SBB local interfaces:

```
public interface ControlDelegate extends SbbLocalObject{
       public void startWithActivity(ActivityContextInterface aci,
                       ControlCoordinator coordinator)
                throws UnacceptableActivityException;
}

public interface ControlCoordinator extends SbbLocalObject{
       public void doneWithActivity(ActivityContextInterface aci,
                       ControlDelegate delegate)
               throws UnknownDelegateException;
}
```

Chapter 6 The SBB Abstract Class

Changed in 1.1: The SBB Abstract Class has been updated to allow an additional method signature for both event handler and fire event methods. The CMP contract has been tightened for Java Object types, and several SLEE types have been added as permissible CMP field types in order to ease common programming tasks.

The SBB abstract class of the SBB component contains the event processing logic and synchronously invoked logic of the SBB component. The SBB Developer must implement an SBB abstract class for every SBB component. An instance of the SLEE generated class that extends the SBB abstract class is known as an SBB object. Each SBB object has a life cycle (see Section 6.2).

SBB objects may be pooled. During the lifetime of an SBB object, it may be assigned to different SBB entities. When the SBB object is assigned to an SBB entity, it can receive events destined for the SBB entity and can manipulate the persistent state of the SBB entity. It can also access the relationships of the SBB entity.

6.1 Overview

The SBB Developer implements the SBB abstract class. The requirements for an SBB abstract class is as follows:

- The SBB abstract class must be defined in a named package, i.e. the class must have a `package` declaration. *(Added in 1.1)*
- The class must implement, directly or indirectly, the `javax.slee.Sbb` interface.
- The class must be defined as `public` and must be `abstract`.
- The class must not define the `finalize` method.

The SBB Developer defines the following concrete and abstract methods in the SBB abstract class.

6.1.1 Concrete methods

These methods contain the application logic of the SBB component. These methods include:

- A public constructor.
  The SBB Developer must provide a public constructor that takes no arguments.
- Life cycle methods. *(Clarified in 1.1)*
  Each SBB object has a life cycle. The SLEE invokes the life cycle methods of the SBB object to make the SBB object aware of its life cycle state. The SLEE invokes a life cycle method before the SLEE moves the SBB object from one life cycle state to another life cycle state. An SBB entity is assigned to an SBB object when the SBB object is in the Ready state. The life cycle states and life cycle methods are described in Section 6.2 and Section 6.3, respectively. The base `Sbb` interface declares the method signatures of these methods and these methods must be implemented in the SBB abstract class.
- Event handler methods.
  The SBB Developer implements an event handler method for each event type that may be received by the SBB. Each event handler method contains the application logic for processing events of a specific event type. The event handler methods are described in Section 8.5.2.
- Local interface methods.
  If the SBB Developer defines an SBB local interface, i.e. an interface that extends the `SbbLocalObject` interface, then the SBB Developer must implement each SBB Developer defined local interface method in the SBB abstract class (see Section 5.6).
- An `sbbExceptionThown` callback method. *(Clarified in 1.1)*
  The SBB Developer must implement the `sbbExceptionThrown` callback method defined in

the base `Sbb` interface. The SLEE invokes this method after a SLEE originated invocation of a transactional method of an SBB object returns by throwing a `RuntimeException` (see Section 6.9.3).

- An `sbbRolledBack` callback method.
  The SBB Developer must implement the `sbbRolledBack` callback method defined in the base `Sbb` interface. The SLEE invokes this method after the SLEE rolls back a transaction that was used to invoke a transactional method of an SBB Object (see Section 6.10.1).
- Initial event selector methods. *(Clarified in 1.1)*
  If the SBB Developer defines an initial event selector method for an initial event type, the SLEE invokes this method for each event of the event type. The return value of this method invocation determines whether or not a new root SBB entity will be instantiated to process this event (see Section 8.6.4).

6.1.1.1 At-most-once invocation semantics

All the above methods have at-most-once invocation semantics. If an invocation of one of these methods fails, the SLEE must not re-invoke the method. For example, the SLEE must not re-invoke a method if the method returns by throwing a checked or `RuntimeException`, or if the transaction that the method was invoked within does not commit.

6.1.2 Abstract methods

These methods are not defined in the base `Sbb` interface because the signatures of these methods are dependent on the specific SBB. The SLEE provides the implementation of these methods when the SBB is installed into the SLEE as part of a deployable unit. The SLEE uses introspection of the SBB abstract class and the definitions in the SBB deployment descriptor to determine and generate the appropriate logic to interface the SBB to the SLEE. These methods include:

- Fire event methods.
  The SBB Developer defines an abstract method for each event type that may be fired by the SBB. For each fire event method, the event type fired by the fire event method determines its method name and the Java type of its event object parameter. The SLEE provided implementation of a fire event method contains the logic to pass the event being fired to the subsystem within the SLEE that performs event routing. The fire event methods are described in Section 8.5.1.
- CMP field get and set accessor methods.
  The SBB Developer uses CMP fields to access the per-instance persistent state (i.e. the SBB entity). CMP fields allow the SLEE to manage the data stored in these fields. For example an implementation could make the state highly available by replicating data to other nodes in a distributed SLEE or could store the data in a persistent stable store. The SBB Developer declares a set of abstract get and set accessor methods for each CMP field. The CMP field being accessed determines the name of the accessor methods and the Java type of the CMP field determines the return type of the get accessor method and the input argument type of the set accessor method. The SLEE provided implementation of a CMP field accessor method contains the glue logic to access the data maintained by the subsystem within the SLEE that manages data persistence. See Section 6.5 for a more detailed description of CMP fields and their accessor methods.
- Get child relation methods.
  The SBB Developer declares a get child relation method for each child relation of the SBB. An SBB entity uses a get child relation method to get access to a `ChildRelation` object. The SBB entity invokes the `ChildRelation` object to create child SBB entities of the child relation. The SBB entity can also enumerate the child SBB entities of the child relation (See Section 6.8).
- Get Profile CMP methods.
  Deprecated in 1.1: The get Profile CMP methods have been deprecated in the 1.1 specification. SBB Developers are encouraged to use the new Profile Table (see Section 10.8) and Profile Local

interfaces (see Section 10.7).
The SBB Developer declares a get Profile CMP method for each Profile CMP interface that the SBB may access. A get Profile CMP method takes a Profile identifier argument and returns an object that implements the declared Profile CMP interface (See Section 6.7).

- An `asSbbActivityContextInterface` method.
  If the SBB Developer defines an SBB Activity Context Interface, then the SBB abstract class must declare an `asSbbActivityContextInterface` method. The SBB uses this method to narrow an object that implements the generic `ActivityContextInterface` to an object that implements the SBB Activity Context Interface so that the SBB can access the Activity Context attributes defined in the SBB Activity Context Interface (see Section 7.7.2).
- Get SBB Usage Parameter methods.
  If the SBB Developer defines an SBB Usage Parameters interface, then the SBB abstract class should declare at least one of the two get SBB Usage Parameters methods. Each of these methods return an object that implements the SBB Usage Parameters interface of the SBB. This allows the SBB to manipulate the usage parameters named by the SBB (see Section 11.4.1).

6.2 SBB object life cycle

An SBB object can be in one of the following three states (see Figure 5).

- Does Not Exist state.
  The SBB object does not exist. It may not have been created or it may have been deleted.
- Pooled state.
  The SBB object exists but is not assigned to any particular SBB entity.
- Ready state.
  The SBB object in the Ready state is assigned to an SBB entity. It is ready to receive events through its event handler methods, local method invocations, and various callback method invocations.

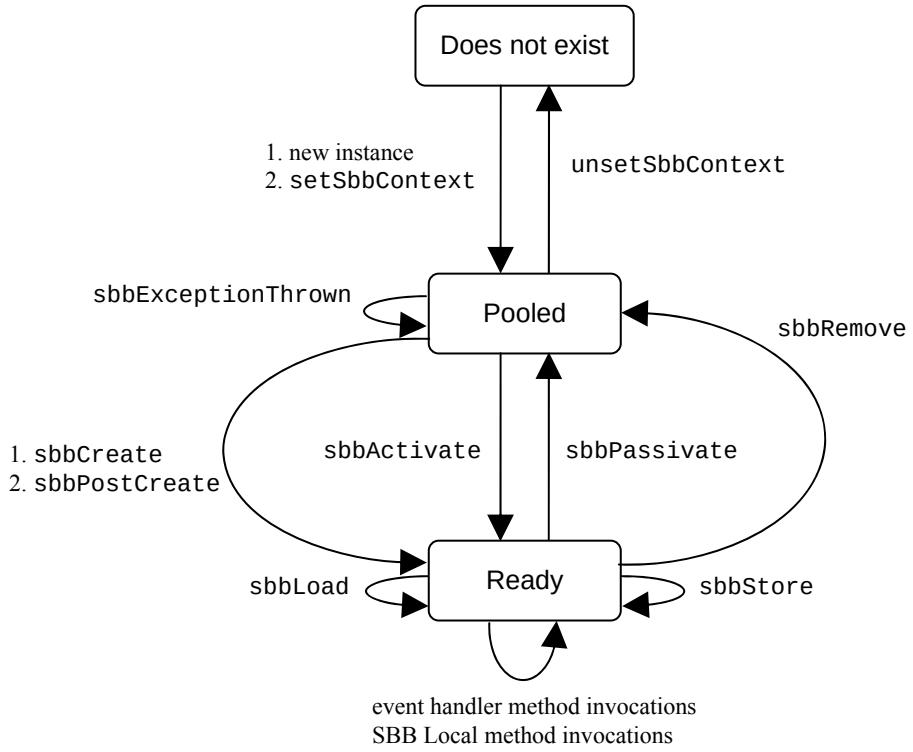

Figure 5 Life cycle of an SBB object

The following steps describe the life cycle of an SBB object:

- An SBB object's life cycle starts when the SLEE creates the object using `newInstance`. The SLEE then invokes the `setSbbContext` method to pass the SBB object an `SbbContext` object. The `SbbContext` object allows the SBB object to invoke functions provided by the SLEE.
- The SBB object enters the pool of available SBB objects. Each SBB component has its own pool. While the SBB object is in the available pool, the SBB object is not associated with any particular SBB entity. All SBB objects in the pool are considered equivalent, and therefore any SBB object can be assigned by the SLEE to an SBB entity during the transition to the Ready state.
- An SBB object transitions from the Pooled state to the Ready state when the SLEE selects that SBB object to process an event or to service a logic object invocation. There are two possible transitions from the Pooled state to the Ready state: through the `sbbCreate` and `sbbPostCreate` methods, or through the `sbbActivate` method. The SLEE invokes the `sbbCreate` and `sbbPostCreate` methods when the SBB object is assigned to a new SBB entity that has just been created explicitly by an invocation of the `create` method on a `ChildRelation` object or implicitly by the SLEE to process an initial event. The SLEE invokes the `sbbActivate` method on an SBB object when an SBB object needs to be activated to receive a method invocation on an existing SBB entity. This occurs when there is no existing SBB object in the Ready state assigned to the SBB entity available to receive the invocation. *(Clarified in 1.1)*
- When an SBB object is in the Ready state, the SBB object is associated with a specific SBB entity. While the SBB object is in the Ready state, the SLEE can synchronize the transient state held in the SBB object with the persistent state of the SBB entity whenever it determines the need to, by invoking the `sbbLoad` and `sbbStore` methods zero or more times. Event handler, local interface, exception callback, and transaction rolled back callback methods can be invoked on the SBB object zero or more times. Invocations of the `sbbLoad` and `sbbStore` methods can be arbitrarily mixed with invocations of these methods subject to the SBB object lifecycle.
- The SLEE can choose to passivate an SBB object. Passivating an SBB object disassociates the SBB object from the SBB entity it is currently assigned to. This allows the SBB object to be reused and assigned to a different SBB entity, for example. To passivate an SBB object, the SLEE first invokes the `sbbStore` method to allow the SBB object to prepare itself for the synchronization of the SBB entity's persistent state with the SBB object's transient state, and then the SLEE invokes the `sbbPassivate` method to return the SBB object to the Pooled state. *(Clarified in 1.1)*
- Eventually, the SLEE will transition the SBB object to the Pooled state. There are three possible normal transitions from the Ready state to the Pooled state: through the `sbbPassivate` method, through the `sbbRemove` method, and because of a transaction rollback for `sbbCreate` or `sbbPostCreate` (not shown in above SBB object life cycle state diagram). The SLEE invokes the `sbbPassivate` method when the SLEE wants to disassociate the SBB object from the SBB entity without removing the SBB entity. The SLEE invokes the `sbbRemove` method when the SLEE wants to remove the SBB entity (i.e. when the SBB entity is removed as part of a cascading removal of an SBB entity sub-tree). If `sbbCreate` and `sbbPostCreate` are called and successfully return without throwing a `RuntimeException` or `CreateException`, and subsequently the enclosing transaction rolls back, the SLEE will transition the SBB object to the Pooled state by invoking the `sbbPassivate` method.
- When the SBB object is put back into the pool, it is no longer associated with the SBB entity. The SLEE can assign the SBB object to any SBB entity of the same SBB component.
- The SLEE may release its references to an SBB object in the pool, allowing it to be garbage collected, after calling the `unsetSbbContext` method on the SBB object.

Notes:

1. The `SbbContext` object passed by the SLEE to the SBB object in the `setSbbContext` method is not an object that contains static information. For example, the result of the `getSbbLocalObject` method might be different each time an SBB object moves from the Pooled state to the Ready state, and the result of the `getActivities` method may be different in different event handler method invocations.
2. An SBB object is only ever associated with one Service. This means the `getService` and `getSbb` methods of an `SbbContext` object always returns the same result during the lifetime of the SBB object. *(Added in 1.1)*
3. A `RuntimeException` thrown from any method of an SBB object (including the event handler methods, the local interface methods, and the life cycle callbacks invoked by the SLEE) results in the transition to the Does Not Exist state after the appropriate exception handler methods have been invoked (see Sections 9.12.1 and 9.12.2). In the case of a SLEE originated invocation (e.g. an event handler method throwing a `RuntimeException`) the SLEE will not invoke any method except the `sbbExceptionThrown` method on the SBB object after a `RuntimeException` has been caught. The corresponding SBB entity continues to exist. The SBB entity can continue to receive events and synchronous invocations because the SLEE can use a different SBB object to process events and synchronous invocations that should be sent to the SBB entity. Exception handling is described further in Section 6.9.
4. A SLEE implementation is not required to maintain a pool of SBB objects in the Pooled state. The pooling approach is an example of a possible implementation, but it is not the required implementation. Whether the SLEE implementation uses a pool or not has no bearing on the SBB abstract class coding style. Regardless of whether or not a pool is used, the SLEE implementation must still follow the same SBB object lifecycle transitions, and invoke the same lifecycle callbacks at the appropriate lifecycle transitions, as specified above. *(Clarified in 1.1)*

6.3 SBB abstract class life cycle methods

The SBB Developer is responsible for implementing the following life cycle methods in the SBB abstract class:

6.3.1 `setSbbContext` method

The `Sbb` interface defines the method signature of the `setSbbContext` method as follows:

```
public void setSbbContext(SbbContext context);
```

The SLEE invokes this method after a new instance of the SBB abstract class is created. It uses this method to pass an `SbbContext` object to the SBB object. If the SBB object needs to use the `SbbContext` object during its lifetime, it should keep a reference to the `SbbContext` object in an instance variable.

During this method, an SBB entity has not been assigned to the SBB object. The SBB object can take advantage of this method to allocate and initialize state or connect to resources that are to be held by the SBB object during its lifetime. Such state and resources cannot be specific to an SBB entity because the SBB object might be reused during its lifetime to serve multiple SBB entities.

- The `setSbbContext` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The SBB object must not attempt to access its persistent state using the CMP field accessor methods during this method
- This method is invoked with an unspecified transaction context (see Section 9.7.1 for details on how the SLEE executes methods with an unspecified transaction context).

6.3.2 `unsetSbbContext` method

The `Sbb` interface defines the method signature of the `unsetSbbContext` method as follows:

```
public void unsetSbbContext();
```

The SLEE invokes this method before terminating the life of the SBB object. During this method, an SBB entity is not assigned to the SBB object. The SBB object can take advantage of this method to free state or resources that are held by the SBB object. These state and resources typically had been allocated by the `setSbbContext` method.

- The `unsetSbbContext` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The SBB object must not attempt to access its persistent state using the CMP field accessor methods during this method.
- This method is invoked with an unspecified transaction context.

6.3.3 `sbbCreate` method

The `Sbb` interface defines the method signature of the `sbbCreate` method as follows:

```
public void sbbCreate() throws javax.slee.CreateException;
```

The SLEE invokes this method on an SBB object *before* the SLEE creates a new SBB entity in response to an initial event or an invocation of the `create` method on a `ChildRelation` object. This method should initialize the SBB object using the CMP field get and set accessor methods, such that when this method returns, the persistent representation of the SBB entity can be created.

- The `sbbCreate` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- This method should throw a `javax.slee.CreateException` when there is an application level problem (rather than SLEE or system level problem). If this method throws this exception, then the SLEE will not create the SBB entity. The SLEE will also propagate the `CreateException` unchanged to the caller that requested the creation of the SBB entity. The caller may be the SLEE or an SBB object. The `throws` clause is mandatory.
- The SLEE guarantees that the values that will be initially returned by the SBB object's CMP field get methods will be the Java language defaults (e.g. 0 for integer, `null` for object references).
- The SLEE invokes this method with a transaction context (see Section 9.6). After this method returns, and in the same transaction context, the SLEE creates the persistent representation of the SBB entity.
- Child SBB entities should not be created in this method. This method is invoked before the persistent representation of the new SBB entity is created so the SLEE may not be able to maintain the relation between the new SBB entity and its child SBB entities. Child SBB entities should instead be created in the `sbbPostCreate` method or in event handling or local invocation methods.

6.3.4 `sbbPostCreate` method

The `Sbb` interface defines the method signature of the `sbbPostCreate` method as follows:

```
public void sbbPostCreate() throws javax.slee.CreateException;
```

The SLEE invokes this method on an SBB object *after* the SLEE creates a new SBB entity. The SLEE invokes this method after the persistent representation of the SBB entity has been created and the SBB object is assigned to the created SBB entity. This method gives the SBB object a chance to initialize additional transient state and acquire additional resources that it needs while it is in the Ready state.

- The `sbbPostCreate` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- This method may throw a `javax.slee.CreateException` when there is an application level problem (rather than SLEE or system level problem). The SLEE will propagate the `Cre-`

`ateException` unchanged to the caller that requested the creation of the SBB entity. The caller may be the SLEE or an SBB object. The `throws` clause is optional.

- The SLEE invokes this method with the transaction context (see Section 9.6) used to invoke the `sbbCreate` method.
- The SBB entity enters the Ready state when `sbbPostCreate` returns normally. If `sbbPost-Create` returns by throwing an exception, the SBB entity does not become Ready.

6.3.5 `sbbActivate` method

```
public void sbbActivate();
```

The SLEE invokes this method on an SBB object in the Pooled state when the SLEE needs to assign it to a specific SBB entity. This method gives the SBB object a chance to initialize additional transient state and acquire additional resources that it needs while it is in the Ready state. *(Clarified in 1.1)*

- The `sbbActivate` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The SBB object must not attempt to access its persistent state using the CMP field accessor methods during this method.
- This method executes with an unspecified transaction context.

6.3.6 `sbbPassivate` method

```
public void sbbPassivate();
```

The SLEE invokes this method on an SBB object when the SLEE decides to disassociate the SBB object from the SBB entity, and to put the SBB object back into the pool of available SBB objects. This method gives the SBB object the chance to release any state or resources that should not be held while the SBB object is in the pool. These state and resources typically had been allocated during the `sbbActivate` method.

- The `sbbPassivate` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The SBB object must not attempt to access its persistent state using the CMP field accessor methods during this method.
- This method executes with an unspecified transaction context.

6.3.7 `sbbRemove` method

```
public void sbbRemove();
```

The SLEE invokes the `sbbRemove` method on an SBB object before the SLEE removes the SBB entity assigned to the SBB object. The SLEE removes an SBB entity when the SBB sub-entity tree or SBB entity tree that the SBB entity belongs to is removed explicitly by an invocation of the `remove` method on an SBB local object or implicitly by the SLEE when the attachment count of root SBB entity decrements to zero. The SBB object is in the Ready state when `sbbRemove` is invoked and it will enter the Pooled state when the method completes. The SBB Developer can use this method to implement any actions that must be done before the SBB entity's persistent representation is removed.

- The `sbbRemove` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The SLEE synchronizes the SBB object's state before it invokes the `sbbRemove` method. This means that the CMP state of the SBB object at the beginning of this method is the same as it would be at the beginning of an event handler method or a local interface method.

- This method is invoked with a transaction context. After this method returns, and in the same transaction context, the SLEE executes the internal operations that delete the persistent representation of the SBB entity.
- Since the SBB object will be entered into the pool of available SBB objects, the state of the SBB object at the end of this method must be equivalent to the state of a passivated SBB object. This means that the SBB object must free any state and release any resource that it would normally release in the `sbbPassivate` method.

6.3.8 `sbbLoad` method

```
public void sbbLoad();
```

The SLEE calls this method to synchronize the state of an SBB object with its assigned SBB entity's persistent state. The SBB Developer can assume that the persistent state of the SBB entity the SBB object is assigned to has been loaded just before this method is invoked. It is the responsibility of the SBB Developer to use this method to re-compute or initialize the values of any transient instance variables in the SBB object that depend on the SBB entity's persistent state. In general, any transient state that depends on the persistent state of an SBB entity should be recalculated in this method. The SBB Developer can use this method, for instance, to perform some computation on the values returned by the CMP field accessor methods, such as converting text fields to more convenient objects or binary representations. *(Clarified in 1.1)*

- The `sbbLoad` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The SLEE invokes this method within a transaction context.

6.3.9 `sbbStore` method

```
public void sbbStore();
```

The SLEE calls this method to synchronize the SBB entity's persistent state with the state of the SBB object. The SBB Developer should use this method to update the SBB entity using the CMP field accessor methods before its persistent state is synchronized. For example, this method may perform conversion of object or binary data representations to text. The SBB Developer can assume that after this method returns, the persistent state is synchronized. *(Clarified in 1.1)*

- The `sbbStore` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The SLEE invokes this method within a transaction context.

6.4 `SbbContext` interface

The SLEE provides each SBB object with an `SbbContext` object. The `SbbContext` object gives the SBB object access to the SBB object's context maintained by the SLEE, allows the SBB object to invoke functions provided by the SLEE, and obtain information about the SBB entity assigned to the SBB object.

An `SbbContext` object is associated with one Service and the associated Service does not change during the lifetime of that `SbbContext` object. *(Added in 1.1)*

The `SbbContext` object implements the `SbbContext` interface. The `SbbContext` interface declares the following methods:

- Methods to access information determined at runtime.
  - The `getSbbLocalObject` method returns an SBB local object that represents the SBB entity assigned to the SBB object.
  - The `getService` operation returns a `ServiceID` object that encapsulates the component identity of the Service related to the SBB entity tree the SBB entity is a part of.

  - The `getSbb` method returns an `SbbID` object that encapsulates the component identity of the SBB.
  - The `getTracer` method returns a tracer for the SBB. The notification source used by the tracer is a `javax.slee.management.SbbNotification` which encapsulates the component identity of the SBB and the Service that the SBB entity is executing on behalf of. *(Added in 1.1)*

- Activity Context method.
  This methods is discussed in greater detail in Section 7.7.1.
  - The `getActivities` method.
    This method returns an Activity Context Interface object for each of the Activity Contexts that the SBB entity assigned to the SBB object of the `SbbContext` object is attached to.
- Event mask methods.
  These methods are described in Section 8.5.3.
  - The `maskEvent` method.
    This method masks event types that the SBB entity assigned to the SBB object of the `SbbContext` object no longer wishes to receive from an Activity Context.
  - The `getEventMask` method.
    This returns the set of masked event types for an Activity Context that SBB entity assigned to the SBB object of the `SbbContext` object is attached to.
- Transaction methods.
  These methods are described in Section 6.10.
  - The `setRollbackOnly` method.
    The SBB Developer uses this method to mark the transaction of the current method invocation for rollback.
  - The `getRollbackOnly` method.
    The SBB Developer uses this method to determine if the transaction of the current method invocation has been marked for rollback.

6.4.1 `SbbContext` interface `getSbbLocalObject` method

This method returns an SBB local object that represents the SBB entity assigned to the SBB object of the `SbbContext` object.

The method signature for the `getSbbLocalObject` method is as follows:

```
SbbLocalObject getSbbLocalObject()
       throws  TransactionRequiredLocalException,
               IllegalStateException,
               SLEEException;
```

- The object returned is an object that implements the SBB local interface of the SBB entity. The SBB object uses Java typecast to narrow the returned object reference (from `SbbLocalObject`) to the SBB local interface of the SBB entity.
- The SBB object of the `SbbContext` object must have an assigned SBB entity when it invokes this method. Otherwise, this method throws a `java.lang.IllegalStateException`.
- It is a mandatory transactional method (see Section 9.6.1). Hence, the SBB object must invoke this method within a transaction. It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

6.4.2 SbbContext interface getService method

This method returns the Service component identifier of the Service for which the SBB object owning the SbbContext object was instantiated for.

The method signature for the getService method is as follows:

```
ServiceID getService()
       throws  SLEEException;
```

- This method is a non-transactional method. An SBB object may invoke this method at any time to get the ServiceID object for the SBB.
- It throws a javax.slee.SLEEException if the requested operation cannot be performed due to a system-level failure.

6.4.3 SbbContext interface getSbb method

This method returns the SBB component identifier of the SBB object for which the SbbContext object was instantiated for.

The method signature for the getSbb method is as follows:

```
SbbID getSbb()
       throws  SLEEException;
```

- This method is a non-transactional method. An SBB object may invoke this method at any time to get the SbbID object for the SBB.
- It throws a javax.slee.SLEEException if the requested operation cannot be performed due to a system-level failure.

6.4.4 SbbContext interface getTracer method

Added in 1.1.

This method returns a tracer for the SBB with the specified tracer name. The notification source used by the tracer is a javax.slee.management.SbbNotification which encapsulates the component identity of the SBB and the Service that the SBB entity is executing on behalf of (as returned by the getSbb and getService methods of SbbContext). Refer to the Trace Facility in Section 13.3 for a complete discussion on tracers and tracer names. Trace notifications generated by a tracer obtained using this method are of the type javax.slee.management.SbbNotification.TRACE_NOTIFICATION_TYPE.

The method signature for the getTracer method is as follows:

```
Tracer getTracer(String tracerName)
       throws  NullPointerException,
               IllegalArgumentException,
               SLEEException;
```

- This method is a non-transactional method. An SBB object may invoke this method at any time to get a tracer object for the SBB.
- It throws a NullPointerException if the tracerName argument is null.
- It throws an IllegalArgumentException if tracerName is an invalid name. Name components within a tracer name must have at least one character. For example “com.mycompany” is a valid tracer name, whereas “com..mycompany” is not.
- A SLEEException is thrown if a Tracer object could not be obtained due to a system-level failure.

6.5 Container Managed Persistence (CMP)

An SBB entity defines its persistent state as a set of CMP fields.

6.5.1 CMP field rules

The SBB Developer must observe the following rules when defining the CMP fields of an SBB:

- From the perspective of the SBB Developer, the CMP fields are *virtual* fields only, and are accessed through get and set accessor methods. The SLEE supplies the implementation of the CMP fields. The CMP fields must not be defined in the SBB abstract class (i.e. as an instance variable in the class.) *(Clarified in 1.1)*
- The CMP fields must be specified in the SBB's deployment descriptor using the `cmp-field` elements respectively. The names of these fields must be valid Java identifiers and must begin with a lowercase letter, as determined by `java.lang.Character.isLowerCase`.
- The SBB Developer must declare the accessor methods for the CMP fields as get and set methods, using the JavaBeans conventions. The SLEE supplies the implementation of the accessor methods.
- The accessor methods must be `public`, must be `abstract`, and must bear the name of the CMP field that is specified in the `cmp-field` deployment descriptor element, and in which the first letter of the name of the CMP field has been uppercased and prefixed by "get" or "set".
- The Java types of the CMP fields are restricted to Java primitive types, Java serializable types, SBB local interfaces (i.e., `SbbLocalObject` and the types derived from `SbbLocalObject`), Activity Context Interface interfaces (i.e., `ActivityContextInterface` and the types derived from `ActivityContextInterface`), Event Context interfaces (i.e., `Event-Context`), and Profile Local interfaces (i.e. `ProfileLocalObject` and the types derived from `ProfileLocalObject`).
  Since the SLEE specification does not require the SBB local interfaces, Activity Context Interface interfaces, Event Context interfaces, and Profile Local interfaces to be serializable, non-primitive CMP field types cannot contain these types. For example, neither a composite object that contains one or more SBB local interface member types nor an SBB local interface array are valid CMP field types.
  (Changed in 1.1: Added Activity Context Interface interfaces, Event Context interfaces, and Profile Local interfaces to permitted types.)
- CMP fields that hold dependent value classes. *(Clarified in 1.1)*
  A dependent value class is a concrete class. A dependent value class may be a legacy class that the SBB Developer wishes to use internally within an SBB with CMP fields, and/or it may be a class that the SBB Developer chooses to expose through the SBB local interface of the SBB. A dependent value class can be the value of a CMP field.
  - The get accessor method for a CMP field that corresponds to a dependent value class returns a copy of the dependent value class instance.
  - The assignment of a dependent value class value to a CMP field using the set accessor method causes the value to be copied to the target CMP field.
  - A dependent value class must be serializable.
- CMP fields that hold SBB entity references. *(Changed in 1.1: `sbb-alias-ref` is now optional)*
  - Logically, setting an SBB local object into a CMP field stores a reference to the SBB entity represented by the SBB local object. The SLEE maintains referential integrity for SBB entity references stored in CMP fields. If the target SBB entity of a reference is deleted, the SLEE must in future return `null` from any CMP field accessor for a CMP field that referenced the target SBB entity.
  - The Java type of a CMP field that can hold an SBB local object reference must be `SbbLocalObject` or a type derived from `SbbLocalObject`. If an `sbb-alias-ref` element is present in the deployment descriptor for the CMP field then the Java type of the CMP field must be the same type or a base type of the SBB local interface of the SBB component specified by the `sbb-alias-ref` element.

  - The declaration of an `sbb-alias-ref` element in the deployment descriptor for a CMP field that stores an SBB entity reference is optional. If an `sbb-alias-ref` element is declared, only SBB local objects of the SBB component specified by the `sbb-alias-ref` element may be stored in the CMP field. If an `sbb-alias-ref` element is not present, any SBB with an SBB local interface that is assignable to the CMP field type can be stored in the CMP field.
  - Only `null` and valid SBB local objects can be passed to the set accessor method. The only valid SBB local objects are the SLEE implemented and provided SBB local objects. The SLEE implemented set accessor method must throw the `java.lang.IllegalArgumentException` if an invalid SBB local object is passed to the set accessor method.
    An example of an invalid SBB local object is an object instantiated from a class implemented by an SBB Developer regardless of whether it implements an SBB local interface.
  - If the `sbb-alias-ref` element is present, only `null` and references to valid SBB entities of the SBB component identified by the `sbb-alias-ref` element can be stored in the CMP field. The SLEE implemented set accessor method must throw the `java.lang.IllegalArgumentException` if any other SBB entity reference, or an invalid reference to an SBB entity of the SBB component identified by the `sbb-alias-ref` element, is passed to the set accessor method.

- CMP fields that hold Activity Context references. *(Added in 1.1)*
  - Logically, setting an Activity Context Interface object into a CMP field stores a reference to the Activity Context represented by the Activity Context Interface object. The SLEE maintains referential integrity for Activity Context references stored in CMP fields. If the target Activity Context of a reference no longer exists (see Section 7.3.3), the SLEE must in future return `null` from any CMP field accessor for a CMP field that referenced the Activity Context.
    Storing a reference to an Activity Context through a CMP field of an SBB entity is different from attaching the SBB entity to the Activity Context in that it does not affect on event delivery to the SBB entity (e.g.. does not make the SBB entity eligible to receive events fired on the Activity Context) and it does not increase the attachment count of the Activity Context.
  - The Java type of a CMP field that can hold an Activity Context reference must be the same type or a base type of the SBB Activity Context Interface of the SBB component.
  - Only `null` and valid Activity Context Interface objects can be passed to the set accessor method. The only valid Activity Context Interface objects are the SLEE implemented and provided Activity Context Interface objects. The SLEE implemented set accessor method must throw the `java.lang.IllegalArgumentException` if an invalid Activity Context Interface object is passed to the set accessor method.
    An example of an invalid Activity Context Interface object is an object instantiated from a class implemented by an SBB Developer regardless of whether or not it implements an Activity Context Interface interface.

- CMP fields that hold Event Context references. *(Added in 1.1)*
  - Logically, setting an Event Context into a CMP field stores a reference to the context of an event associated to the Event Context object. The SLEE maintains referential integrity for Event Context references stored in CMP fields. If the target Event Context of a reference is deleted, the SLEE must in future return `null` from any CMP field accessor for a CMP field that referenced the target Event Context.
  - The Java type of a CMP field that can hold an Event Context reference must be `EventContext`.

  - Only `null` and valid Event Context objects can be passed to the set accessor method. The only valid Event Context objects are the SLEE implemented and provided Event Context objects. The SLEE implemented set accessor method must throw the `java.lang.IllegalArgumentException` if an invalid Event Context object is passed to the set accessor method.
    An example of an invalid Event Context object is an object instantiated from a class implemented by an SBB Developer regardless of whether it implements the Event Context interface.

- CMP fields that hold Profile references. *(Added in 1.1)*
  - Logically, setting a Profile Local object into a CMP field stores a reference to the Profile represented by the Profile Local object. The SLEE maintains referential integrity for Profile references stored in CMP fields. If the target Profile of a reference is deleted, the SLEE must in future return `null` from any CMP field accessor for a CMP field that referenced the target Profile.
  - The Java type of a CMP field that can hold a Profile reference must be `ProfileLocalObject` or one of its derived types.
  - Only `null` and valid Profile Local objects can be passed to the set accessor method. The only valid Profile Local objects are the SLEE implemented and provided Profile Local objects. The SLEE implemented set accessor method must throw the `java.lang.IllegalArgumentException` if an invalid Profile Local object is passed to the set accessor method.
    An example of an invalid Profile Local object is an object instantiated from a class implemented by an SBB Developer regardless of whether it implements the Profile Local interface.
- The CMP field accessor methods are mandatory transactional methods (see Section 9.6.1). They throw a `javax.slee.TransactionRequiredLocalException` if they are invoked without a valid transaction context.
- These methods may also throw a `javax.slee.SLEEException` if the operation cannot be completed due to a system-level failure.
- Neither the get accessor method nor the set accessor method can include a `throws` clause.

6.5.2 CMP field example

Changed in 1.1: Added Activity Context Interface, Event Context, and Profile Local Object types to the example.

The following example illustrates two CMP fields and their get and set accessor methods.

The deployment descriptor for the example Foo SBB is as follows:

```
<sbb>
        …
        <description> FooSbb … </description>
        <sbb-name> Foo SBB </sbb-name>
        <sbb-vendor> com.foobar </sbb-vendor>
        <sbb-version> 1.0 </sbb-version>
        <sbb-alias> FooSbb </sbb-alias>
        …
        <sbb-classes>
                <sbb-abstract-class>
                        …
                        <sbb-abstract-class-name>
                                com.foobar.FooSbb
                        </sbb-abstract-class-name>
                        <cmp-field>
                                <description>
                                        An integer counter.
```

```
                                </description>
                                <cmp-field-name> counter </cmp-field-name>
                        </cmp-field>
                        <cmp-field>
                                <description>
                                        The most recent time that the counter was updated.
                                </description>
                                <cmp-field-name> counterLastUpdated </cmp-field-name>
                        </cmp-field>
                        <cmp-field>
                                <description>
                                        Holds a reference to my peer FooSBB.
                                </description>
                                <cmp-field-name> peerFooSbb </cmp-field-name>
                                <sbb-alias-ref> FooSbb </sbb-alias-ref>
                        </cmp-field>
                        <cmp-field>
                                <description>
                                        Holds a reference to an Activity Context.
                                </description>
                                <cmp-field-name> myActivityContext </cmp-field-name>
                        </cmp-field>
                        <cmp-field>
                                <description>
                                        Holds a reference to an Event Context.
                                </description>
                                <cmp-field-name> myEventContext </cmp-field-name>
                        </cmp-field>
                        <cmp-field>
                                <description>
                                        Holds a reference to a BarProfileLocalObject.
                                </description>
                                <cmp-field-name> myProfileLocal </cmp-field-name>
                        </cmp-field>

                        …
                </sbb-abstract-class>
                …
        </sbb-classes>
        …
</sbb>
```

The Foo SBB abstract class is as follows:

```
package com.foobar;
…
import java.util.Date;
…
public abstract FooSbb implements javax.slee.Sbb {
        …
        /* accessor methods for counter CMP field */
        public abstract int getCounter();
        public abstract void setCounter(int value);

        /* accessor methods for counterLastUpdated CMP field */
        public abstract Date getCounterLastUpdated();
        public abstract void setCounterLastUpdated(Date value);

        /* accessor methods for FooSbbLocalObject CMP field */
        public abstract FooSbbLocalObject getPeerFooSbb();
        public abstract void setPeerFooSbb(FooSbbLocalObject value);

        /* accessor methods for Activty Context CMP field */
        public abstract ActivityContextInterface getMyActivityContext();
        public abstract void setMyActivityContext(ActivityContextInterface aci);

        /* accessor methods for Event Context CMP field */
        public abstract EventContext getMyEventContext();
        public abstract void setMyEventContext(EventContext ctx);
```

```
        /* accessor methods for Profile Local Object CMP field */
        public abstract BarProfileLocalObject getMyProfileLocal();
        oublic abstract void setMyProfileLocal(BarProfileLocalObject);

        …
}
```

6.5.3 Similarities and Differences with respect to EJB 2.0

Clarified in 1.1.

This section is non-normative. The CMP architecture of the SLEE specification is loosely based on the CMP architecture of Enterprise JavaBeans 2.0 specification. For developers who are already familiar with EJB, some of the primary differences are as follow:

- SLEE CMP is used to access the persistent state of the SBB entity. The SBB entity and its relationship to SBB objects is described by the SBB object lifecycle. Unlike EJB entity beans that are always explicitly created and deleted by application code, the SLEE creates root SBB entities to receive events, and deletes SBB entity trees that can no longer receive events. The SLEE automatically creates root SBB entities and their persistent state in response to initial events, and the SLEE automatically deletes SBB entities in an SBB entity tree and their persistent state when the SBB entity tree will no longer receive further events. Furthermore, SBB entities and their persistent state do not have an externally visible primary key.
- Typically, EJB CMP is used to represent or model data in databases or back-end legacy applications. The data modeled by EJB CMP are stored in persistent data stores. On the other hand, SLEE CMP is used to identify data that should be managed by the SLEE. This allows a SLEE to provide various durability semantics, e.g. to replicate the data for redundancy when handling system failures. The SLEE specification does not define how the persistent state of an SBB entity should be managed. It can be replicated to traditional data sources, like EJB, or may be replicated in the memory of one or more other nodes within a distributed SLEE, or not replicated at all.
- CMP in EJB 2.0 defines both CMP fields and Container Managed Relationship (CMR) fields. The SLEE specification does not include CMR.

6.6 Non-persistent state

The SBB Developer may also declare instance variables in the SBB abstract class to maintain non-persistent state. The SBB Developer can use instance variables to store values that depend on the persistent state of the SBB entity, although this use is not encouraged. The SBB Developer should use the `sbbLoad` method to resynchronize the values of any instance variables that depend on the SBB entity's persistent state. In general, any non-persistent state that depends on the persistent state of an SBB entity should be recomputed during the `sbbLoad` method.

An SBB Developer must be careful when using instance variables in an SBB abstract class. For example, instance variables should not be used to store temporary data between invocations of event handler methods, as the SLEE specification makes no guarantees that such invocations would be made on the same SBB object. Instance variables should only be used to store data that exists for the lifetime of the SBB object, for example a reference to an `SbbContext` object , or to cache data that is saved to or restored from CMP fields during the appropriate lifecycle transitions. *(Clarified in 1.1)*

6.7 SBB abstract class get Profile CMP methods

Deprecated in 1.1: The get Profile CMP methods have been deprecated in the 1.1 specification. SBB Developers are encouraged to use the new Profile Table (see Section 10.8) and Profile Local interfaces (see Section 10.7) as get Profile CMP methods may be removed in the future.

The SBB Developer declares a get Profile CMP method for each Profile Specification that the SBB may use and provides a deployment descriptor to specify the Profile Specification of the get Profile CMP method.

The method signature of this method is as follows:

```
public abstract <Profile CMP interface> <get Profile CMP method name>
                (ProfileID profileID)
        throws  UnrecognizedProfileTableNameException,
                UnrecognizedProfileNameException;
```

- The method must be declared as `public` and `abstract`. The SLEE provides the concrete implementation of the get Profile CMP method when the SBB is deployed.
- The method name must not begin with "sbb" or "ejb", and must be a valid Java identifier.
- The Java type of the only input argument must be `ProfileID`.
  When the SBB invokes this method with a `ProfileID` object, the `ProfileID` object should uniquely identify a Profile that is in a Profile Table created from the Profile Specification.
  - It throws a `javax.slee.profile.UnrecognizedProfileTableNameException` if the `ProfileID` object does not identify a Profile Table created from the Profile Specification.
  - If the Profile identifier identifies a Profile Table created from the Profile Specification, then this method throws a `javax.slee.profile.UnrecognizedProfileNameException` if the `ProfileID` object does not identify a Profile within the Profile Table.
- The return type must be the Profile CMP interface of the Profile Specification, or a base interface of the Profile CMP interface of the Profile Specification.
  The SBB uses the returned Profile CMP object to access the attributes of the Profile identified by the input argument.
- The SBB object must have an assigned SBB entity when it invokes this method. Otherwise, this method throws a `java.lang.IllegalStateException`. This exception is not declared in the `throws` clause.
- It is a mandatory transactional method (see Section 9.6.1). Hence, the SBB object must invoke this method within a transaction. It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context. This exception is not declared in the `throws` clause.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure. This exception is not declared in the `throws` clause.

The `get-profile-cmp-method` deployment descriptor element declares the presence of the get Profile CMP method and associates the method with a Profile Specification. This SBB Developer specifies the following aspects of the get Profile CMP method using the `get-profile-cmp-method` element:

- The method name of the get Profile CMP method.
- The Profile Specification of the get Profile CMP method.

6.8 SBB abstract class get child relation methods

For each child relation of the SBB, the SBB Developer must declare an abstract get child relation method in the SBB abstract class and include a `get-child-relation-method` deployment descriptor element in the SBB's deployment descriptor.

The method signature of a get child relation method is as follows:

```
public abstract ChildRelation <get child relation method name>();
```

- The method must be `public` and `abstract`. The SLEE provides the concrete implementation of the get child relation method when the SBB is deployed.
- The method name must not begin with "sbb" or "ejb".

- The method does not have a `throws` clause.
- The return type must be `javax.slee.ChildRelation`. An SBB entity invokes the returned `ChildRelation` object to create child SBB entities and enumerate child SBB entities that belong to this relation (see Section 6.8.1).
- The `ChildRelation` object and its `Iterator` objects are only valid within the same transaction in which the `ChildRelation` object was obtained.
- The name of the method, i.e. *<get child relation method name>*, must be a valid Java identifier.
- The method must not throw any checked exceptions.
- The SBB object must have an assigned SBB entity when it invokes this method. Otherwise, this method throws a `java.lang.IllegalStateException`.
- It is a mandatory transactional method (see Section 9.6.1). Hence, the SBB object must invoke this method within a transaction. It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

The `get-child-relation-method` deployment descriptor element declares the child relation from the SBB to the child SBB. This SBB Developer specifies the following aspects of the child relation using the `get-child-relation-method` element:

- The child SBB of the child relation, i.e. the SBB that plays the child role in this child relation.
- The method name of the get child relation method for this child relation.
- The default event delivery priority of the child relation.

6.8.1 `ChildRelation` interface

A `ChildRelation` object is an instance of a class that implements the `ChildRelation` interface. The SLEE provides the implementation of this class.

The `ChildRelation` interface is as follows:

```
package javax.slee;

public interface ChildRelation extends Collection {
       SbbLocalObject create()
              throws  CreateException,
                      TransactionRequiredLocalException,
                      SLEEException;
}
```

6.8.2 `ChildRelation` interface `create` method

An SBB object that represents an SBB entity of a parent SBB invokes this method to create a new child SBB entity on a child relation.

- The actual object returned by this method is an object that implements the SBB local interface of the child SBB. The parent SBB uses Java typecast to narrow the returned object reference (from `SbbLocalObject`) to the SBB local interface of the child SBB.
- If this method returns by throwing a `javax.slee.CreateException` (or a subclass of `CreateException`), the `CreateException` indicates that an application-level error occurred. If a caller of this method receives this exception, the caller does not know, in general, whether the SBB entity was created but not fully initialized, or not created at all. Also, the caller does not know whether or not the transaction has been marked for rollback. However, the caller

may determine the transaction status by using the `getRollbackOnly` method (see Section 6.10.3).

- This method is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- This method throws a `javax.slee.SLEEException` if the child SBB entity could not be created due to a system-level failure.

6.8.3 `ChildRelation` interface `Collection` methods

A `ChildRelation` object is also a collection that contains a set of SBB local objects that represent each child SBB entity of the child relation.

- The `size` method.
  - This method returns the number of child SBB entities in this child relation.
- The `isEmpty` method.
  - This method returns `true` if there are no child SBB entities in this child relation.
- The `contains` method.
  - This method returns `true` if the SBB entity represented by the SBB local object specified by the input argument is a member of this child relation. If the method argument is not an SBB local object, is an invalid SBB local object, or is an SBB local object whose underlying SBB entity is not a member of this child relation, then this method returns `false`.
- The `containsAll` method.
  - This method returns `true` if all SBB entities represented by the SBB local objects in the collection specified by the input argument are members of this child relation. If the collection contains an object that is not an SBB local object, an SBB local object that is invalid, or an SBB local object whose underlying SBB entity is not a member of this child relation, then this method returns `false`.
- The `toArray` methods.
  - The SLEE must implement these methods by following the contract defined by the `java.util.Collection` interface. The ordering of the elements in the returned arrays is unspecified.
- The add methods: `add` and `addAll`.
  - Any attempts to add to the child relation using these methods result in a `java.lang.UnsupportedOperationException`. For example, the SLEE must throw the `UnsupportedOperationException` when these methods are invoked with arguments that add to the collection.
  - An invocation of an add method that has no effect on the collection, such as invoking the `addAll` method with an empty collection may or may not throw an `UnsupportedOperationException` .
  - The `create` method of the `ChildRelation` interface should be used to create a child SBB entity. This causes the SLEE to automatically add an SBB local object that represents the newly created SBB entity to the collection.
- The remove methods: `clear`, `remove`, `removeAll`, and `retainAll`.
  - These methods may remove SBB entities from the child relation. The input argument specifies which SBB entities will be removed from the child relation or retained in the

child relation by specifying the SBB local object or collection of SBB local objects that represent the SBB entities to be removed or retained.

- Removing an SBB entity from a child relation initiates a cascading removal of the SBB entity tree rooted by the SBB entity, similar to invoking the `remove` method on an SBB local object that represents the SBB entity.

- The `iterator` method and the `Iterator` objects returned by this method.
  - The SLEE must implement the methods of the `Iterator` objects returned by the `iterator` method by following the contract defined by the `java.util.Iterator` interface.
  - The `next` method of the `Iterator` object returns an object that implements the SBB local interface of the child SBB of the child relation. Java typecast can be used to narrow the returned object reference to the SBB local interface of the child SBB.
  - The `remove` method of the `Iterator` object removes the SBB entity that is represented by the last SBB local object returned by the `next` method of the `Iterator` object from the child relation.
    - Removing an SBB entity from a child relation by invoking this `remove` method initiates a cascading removal of the SBB entity tree rooted by the SBB entity, similar to invoking the `remove` method on an SBB local object that represents the SBB entity.
    - The behavior of the `Iterator` object is unspecified if the underlying child relation is modified while the iteration is in progress in any way other than by calling this `remove` method.

- The `ChildRelation` object and its `Iterator` objects are only valid within the same transaction that invoked the get child relation method to obtain the `ChildRelation` object.
- If any of the arguments to one of these methods are `null`, then the method throws a `java.lang.NullPointerException`.

6.8.4 Transactional semantics

The methods of the `ChildRelation` interface have mandatory transactional semantics. Hence, these methods must be invoked within a transaction (see Section 9.6.1).

- If a child SBB entity is created, its effect is only visible within the same transaction, i.e. the `ChildRelation` object will contain an SBB local object that represents the recently created SBB entity.
  - If the transaction commits, the new child SBB entity will be visible to subsequent transactions.
  - If the transaction aborts, the new child SBB entity will cease to exist and will not be visible to subsequent transactions.
- Similarly if an SBB entity is removed, its effect is only visible within the same transaction, i.e. the `ChildRelation` object returned will not contain an SBB local object that represents the removed SBB entity and CMP fields that reference this SBB local object will be set to `null`. The SLEE will mark the transaction for rollback and throw a `javax.slee.TransactionRolledbackLocalException` if an SBB local object that represents the removed SBB entity is invoked.
  - If the transaction commits, the removed child SBB entity will cease to exist permanently and will be invisible to subsequent transactions.
  - If the transaction aborts, the "removed" child SBB is not removed and will continue to be visible to subsequent transactions.

6.9 SBB abstract class exception callback method

The SLEE invokes the `sbbExceptionThrown` callback method to handle `RuntimeException`s thrown by the event handler methods, the transaction rolled back callback method and the logical cascade-removal method of the SBB abstract class. This method is not invoked for `RuntimeException` thrown from SBB local object invocations. The distinction between the two cases is described in Section 9.12. *(Clarified in 1.1)*

A well-written SBB should not throw any `RuntimeException`s from any of its SLEE invoked methods. Instead, the SBB should place the exception handling logic inside a `try { } catch (Throwable)` clause and handle `RuntimeException`s within each invoked method.

6.9.1 `RuntimeException` handling for transactional methods

When a SLEE originated mandatory transactional method is invoked on an SBB object and the invocation returns with a `RuntimeException` thrown, the SLEE performs the following actions:

- The SLEE must log this condition. *(Actions reordered in 1.1)*
- The SLEE marks the transaction of the invocation for rollback.
- The SLEE invokes the `sbbExceptionThrown` method of the same SBB object with the same transaction. The SBB object may in the Pooled state or in the Ready state. For example, if an `sbbCreate` method throws the `RuntimeException`, then the SBB object remains in the Pooled state when the SLEE invokes the `sbbExceptionThrown` method on the SBB object. If an event handler method throws the `RuntimeException`, then the SBB object remains in the Ready state when the SLEE invokes the `sbbExceptionThrown` method on the SBB object.
- The SLEE moves the SBB object to the Does Not Exist state.
- If the mandatory transactional method was invoked as a result of an SBB object invoking a method on an SBB local object (i.e. non-SLEE originated invocation as defined Section 9.8), then the SLEE throws a `javax.slee.TransactionRolledBackLocalException` to the caller that invoked the SBB local object.
- If the `sbbExceptionThrown` method of the SBB object returns with another `RuntimeException` thrown, the SLEE must log this condition. The `sbbExceptionThrown` method is not reinvoked in this case.

Section 6.13 summarizes which SBB abstract class methods are mandatory transactional methods. Semantics of mandatory transactional methods are described in Section 9.6.1.

6.9.2 `RuntimeException` handling for non-transactional methods

When the SLEE invokes a non-transactional method of an SBB object and the invocation returns by throwing a `RuntimeException`, the SLEE performs following sequence of operations.

- The SLEE must log this occurrence. *(Actions reordered in 1.1)*
- The SLEE moves the SBB object to the Does Not Exist state.

Section 6.15 summarizes which SBB abstract class methods are non-transactional methods.

6.9.3 SBB abstract class `sbbExceptionThrown` method

The SBB Developer implements the `sbbExceptionThrown` method in the SBB abstract class. The `sbbExceptionThrown` method has the following method signature:

```
public void sbbExceptionThrown(Exception exception,
                               Object event,
                               ActivityContextInterface activity);
```

- The `sbbExceptionThrown` method must be declared as `public` and cannot be `static` , `abstract`, or `final`.
- The `exception` argument is the exception thrown by one of the methods invoked by the SLEE, e.g. a life cycle method, an event handler method, or a local interface method.
- If the method that threw the exception is an event handler method, the `event` and `activity` arguments will be the `event` and `activity` arguments of the event handler method. Otherwise, the `event` and `activity` arguments will be `null`.
- The `sbbExceptionThrown` method is a mandatory transactional method.
- The SLEE does not invoke the `sbbExceptionThrown` method if a non-transactional method invocation returns by throwing a `RuntimeException`.
- If `sbbExceptionThrown` method is invoked on an SBB object in the Ready state, the state of the SBB object remains as it was at the point that the `RuntimeException` was thrown. The SBB object moves to the Does Not Exist state after the `sbbExceptionThrown` method has been invoked. *(Clarified in 1.1)*

6.9.4 SBB abstract class exception callback methods example

```
package com.foobar;
…
import javax.slee.ActivityContextInterface;
import javax.slee.Sbb;
…
public FooSbb implements Sbb … {
        …
        public void sbbExceptionThrown(Exception exception,
                                       Object event,
                                       ActivityContextInterface activity) {
                // this should not happen, generate trace message
                …
                // do additional cleanup
                …
        }
        …
}
```

6.10 SBB abstract class and `RolledBackContext` interface transaction methods

The section describes the transaction related methods of the SBB abstract class and the `SbbContext` interface.

6.10.1 SBB abstract class `sbbRolledBack` callback method

The SLEE invokes the `sbbRolledBack` callback method after a transaction used in a SLEE originated invocation has rolled back (see Section 9.12).

The method signature of the `sbbRolledBack` method is as follows:

```
        public void sbbRolledBack(RolledBackContext context);
```

- The `sbbRolledBack` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- This method is invoked with a new transaction context, i.e. not with the transaction context of the transaction that has been rolled back.
- This method is a mandatory transactional method.
- This method is only invoked on SBB objects in the Ready state.

- If this method returns by throwing an exception, the SLEE processes this exception like the exception has been thrown from an event handler method, i.e. by invoking the `sbbExceptionThrown` method in the same transaction.
- In the case that a `RuntimeException` causes a transaction to be rolled back, the `sbbRolledBack` method is always invoked after the `sbbExceptionThrown` method since the `sbbExceptionThrown` method is invoked in the transaction that will eventually be rolled back. If the `sbbRolledBack` method returns by throwing a `RuntimeException`, then the `sbbExceptionThrown` method will be invoked again in the same transaction that invoked the `sbbRolledBack` method.
- If the transaction that invoked the `sbbRolledBack` method also rolls back, the SLEE must log this condition and not invoke the `sbbRolledBack` method again.
- When the SLEE rolls back a transaction, the SLEE may start zero or more new transactions to deliver transaction rolled back callbacks.
    - If the rolled back transaction does not invoke any SBB entities, i.e. none of the SLEE originated invocations (see Section 9.8.2) of the transaction (see Section 9.8.3) have a target SBB entity, then the SLEE does not start any new transactions to deliver transaction rolled back callbacks.
    - If the rolled back transaction includes one or more SLEE originated invocations (see Section 9.8.2), then the SLEE starts a new transaction to deliver a transaction rolled back callback for each of the SLEE originated invocations in the transaction that has a target SBB entity.

      The SLEE may or may not be successful in delivering a transaction rolled back callback. For example, if the SLEE does not have already have an SBB object in the Ready state that represents the target SBB entity, then SLEE has to invoke the necessary life cycle methods on an SBB object before it can invoke the `sbbRolledBack` method. The SLEE does not invoke the `sbbRolledBack` method if one of these life cycle methods throw a `RuntimeException` and cause the new transaction to be rolled back before the SLEE has an opportunity to invoke the `sbbRolledBack` method.

      For example, the SLEE starts a transaction so that it can invoke an event handler method of a Foo SBB. This event handler method directly invoked by the SLEE synchronously invokes a local interface method of another SBB. The implementation of this local interface method throws a `RuntimeException`. The SLEE marks the transaction for roll back. Eventually, the event handler method returns (whether by throwing an exception or normally). The SLEE eventually starts a new transaction to invoke the `sbbRolledBack` method of a Foo SBB object.

6.10.1.1 `RolledBackContext` interface

The `RolledBackContext` interface is as follows:

```
package javax.slee;

public interface RolledBackContext {
       Object getEvent();
       ActivityContextInterface getActivityContextInterface();
       boolean isRemoveRolledBack();
};
```

- The `getEvent` and `getActivityContextInterface` methods return `null` if the transaction that was rolled back did not deliver an event to an SBB. Otherwise, these methods return the event object and the Activity Context that of the event that should be handled by the transaction that was rolled back, respectively.

- The `isRemovedRolledBack` method returns `true` if the transaction that was rolled back includes a SLEE originated logical cascading removal method invocation (see Section 9.8.1).

  The SLEE originated logical cascading removal method invocation must have been initiated in the transaction that has been rolled back for this method to return `true`.

6.10.2 `SbbContext` interface `setRollbackOnly` method

An SBB object invokes this method within the context of a transaction to mark the transaction for rollback.

The method signature of the `setRollbackOnly` method is as follows:

```
void setRollbackOnly() throws TransactionRequiredLocalException, SLEEException;
```

- It is a mandatory transactional method. If this method is invoked when the SBB object is not invoked within a transaction, then the SLEE will throw a `javax.slee.TransactionRequiredLocalException`.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

6.10.3 `SbbContext` interface `getRollbackOnly` method

An SBB object invokes this method when it is invoked within a transaction to learn if the transaction has been marked for rollback.

The method signature of the `getRollbackOnly` method is as follows:

```
boolean getRollbackOnly() throws TransactionRequiredLocalException, SLEEException;
```

- It is a mandatory transactional method. If this method is invoked when the SBB object is not invoked with a transaction, then the SLEE will throw a `javax.slee.TransactionRequiredLocalException`.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

6.10.4 SBB abstract class and `SbbContext` interface transaction methods example

```
package com.foobar;
…
import javax.slee.Sbb;
import javax.slee.SbbContext;
import javax.slee.RolledBackContext;
…
public FooSbb implements Sbb … {
        …
        private SbbContext sbbContext;
        …
        public void setSbbContext(SbbContext context) {
                …
                sbbContext = context;
                …
        }
        …
        public void onStarted(…) {
                …
                // touch some CMP fields
                …
                // if marked for rollback, do …
                if (sbbContext.getRollbackOnly()) {
                        …
                        return …;
                }
                …
```

```
                try {
                        // do more processing
                } catch (…) {
                        …
                        // something bad happened, do not commit transaction
                        sbbContext.setRollbackOnly();
                        …
                }
                …
        }
        …
        public void sbbRolledBack(RolledBackContext ctx){
                // generate trace message
                …
                // do additional cleanup
                …
        }
        …
}
```

6.11 Non-reentrant and re-entrant SBB components

An SBB Developer can specify that an SBB component is non-reentrant. If an SBB entity of a non-reentrant SBB component is executing in a given transaction context, and another method invocation with the same transaction context arrives for the same SBB entity, the SLEE will throw an exception to the second request. This rule allows the SBB Developer to program the SBB component as single-threaded non-reentrant code.

The functionality of SBB components may require loop backs in the same transaction context. An example of a loop back is when SBB entity A receives a method invocation, A calls SBB entity B, and B calls back A in the same transaction context. The method invoked by the loop back shares the current execution context (which includes the transaction) with the initial invocation of SBB entity A.

If the SBB component is specified as non-reentrant in the deployment descriptor, the SLEE must reject an attempt to re-enter the SBB entity while the SBB entity is executing a method invocation. (This can happen, for example, if the SBB entity has invoked another SBB entity and the other SBB entity tries to make a loop back call.) If the attempt is made to reenter the SBB entity, the SLEE must throw the `javax.slee.SLEEException` to the caller. The SLEE must allow the call if the SBB component's deployment descriptor specifies that the SBB component is re-entrant.

Re-entrant SBB components must be programmed with caution as the SBB Developer must code the SBB abstract class with the anticipation of a loop back call.

SBB components that do not need callbacks should be marked as non-reentrant in the deployment descriptor, allowing the SLEE to detect and prevent illegal concurrent calls.

6.12 Method name restrictions

Non-private (such as public, protected, or package private) methods that are declared by the SBB Developer must not begin with "sbb" or "ejb". A SLEE implementation can use method names that begin with "sbb" when needed without being concerned with possible method name conflicts with SBB Developer declared method names.

This restriction does not apply when the SBB Developer is implementing a required "`sbb<`*XXX*`>`" method declared by the SLEE, such as the life cycle methods declared in the `javax.slee.Sbb` interface. *(Clarified in 1.1)*

6.13 SBB component environment

The component environment of an SBB component provides the runtime environment that is common to all instances of the classes of the SBB component. The SBB component environment is a mechanism that provides the following features:

- Access to Facilities defined by the SLEE specification.
- Customization of the SBB component during assembly or deployment using deployment descriptor elements.
  This allows an SBB Developer and a Service Deployer to customize an SBB component's behavior without the need to access or change the SBB component's source code. The SLEE specification defines SBB deployment descriptor elements that add new context information to the component environment of the SBB component. The SBB Developer and the Service Deployer may use these deployment descriptor elements to provide configuration data that can be determined and fixed during SBB composition or Service deployment. In addition, SBB components may also access external resources that are dependent on the target operational environment. The Service Deployer may use deployment descriptor elements to associate external resources with the SBB component.

The ability to customize the environment of the SBB component at assembly and deployment allows independent software vendors to develop SBB components that are, to a large degree, independent from the operational environment in which the SBB components will be deployed.

6.13.1 SBB component environment as a JNDI naming context

The SLEE implements the SBB component environment, and provides it to the instances of the SBB component classes through the JNDI interfaces. The SBB component environment is used as follows:

- The methods implemented by classes of the SBB component access the component environment using the JNDI interfaces.
- The SLEE provides an implementation of the JNDI naming context that stores the SBB component environment. The SLEE makes the environment naming context available to the instances of the classes of the SBB component at runtime. These instances use the JNDI interfaces to obtain the values of the environment entries.
- The SLEE provides the following environment entries: *(Clarified in 1.1)*
  - An entry for each of the following SLEE Facilities:
    - Timer Facility
    - Alarm Facility
    - Trace Facility
    - Profile Facility
    - Activity Context Naming Facility
  - An entry for each of the following Activity object factories:
    - The Null Activity Factory
    - The Service Activity Factory
  - An entry for each SLEE specification defined Activity Context Interface Factory:
    - The Profile Table Activity Context Interface Factory
    - The Null Activity Context Interface Factory
    - The Service Activity Context Interface Factory
  - An entry for each Resource adaptor type Activity Context Interface Factory binding defined in the SBB's deployment descriptor (see section 6.13.2).
  - An entry for each Resource adaptor entity binding defined in the SBB's deployment descriptor (see section 6.13.3).
- The SBB Developer declares in the SBB deployment descriptor all the environment entries that the SBB component expects Service Deployers to customize during Service assembly and deployment.

Each SBB component has its own set of environment entries. All instances of SBB component classes of the same SBB component share the same environment entries; the environment entries are not shared with other SBB components. Instances of the classes of the SBB component are not allowed to modify the SBB component environment at runtime.

Terminology warning: The SBB's component "environment" should not be confused with the "environment properties" defined in the JNDI documentation.

6.13.1.1 SBB Developer responsibilities

This section describes the SBB Developer's view of the component environment, and defines the accompanying responsibilities.

6.13.1.1.1 Access to SBB component environment

SBB Objects locate the environment naming context using the JNDI interfaces. An SBB Object creates a `javax.naming.InitialContext` object by using the constructor with no arguments, and looks up the environment naming via the `InitialContext` under the name `java:comp/env`. The SBB component environment entries are stored directly in the environment naming context, or in any of its direct or indirect subcontexts.

The following table lists the names of the SLEE specification defined environment entries under `java:comp/env`.

Objects	*Returned Java type*	*Names*
SLEE Facilities	Depends on Facility	slee/facilities/...
Resources	Depends on resource	slee/resources/...

For the environment entries declared by SBB Developers, the value of an environment entry is of the Java type declared by the SBB Developer in the SBB deployment descriptor.

The following example illustrates how environment entries are accessed.

```
package com.foobar;
…
public abstract class FooSbb implements javax.slee.Sbb … {
        …
        public void onNewConnection(…) {
                …
                // Obtain the SBB component environment naming context.
                Context initCtx = new InitialContext();
                Context myEnv = (Context)initCtx.lookup("java:comp/env");

                // Obtain the maximum number of connections
                // configured by the Service Deployer.
                Integer max = (Integer) myEnv.lookup("maxConnections");

                // Obtain the minimum number of connections
                // configured by the Service Deployer.
                Integer min = (Integer) myEnv.lookup("minConnections");

                if (numberConnections > max.intValue() ||
                    numberConnections < min.intValue()) {
                        // terminate connection and return
                        …
                }

                // Get some more environment entries. These environment
                // entries are stored in subcontexts.
                String val1 = (String) myEnv.lookup("foo/name1");
                Boolean val2 = (Boolean) myEnv.lookup("foo/bar/name2");

                // Can also lookup using full pathnames.
                Integer val3 = (Integer)
                initCtx.lookup("java:comp/env/name3");
                Integer val4 = (Integer)
```

```
                initCtx.lookup("java:comp/env/foo/name4");
                …
        }
        …
}
```

6.13.1.1.2 Declaration of environment entries

The SBB Developer must declare all the environment entries accessed from the SBB component's code. The environment entries are declared using the `env-entry` elements in the SBB deployment descriptor.

Each `env-entry` element describes a single environment entry. The `env-entry` element consists of the following sub-elements:

- An optional `description` element that describes the environment entry.
- An `env-entry-name` element that specifies a name relative to the `java:comp/env` context.
- An `env-entry-type` element that specifies the expected Java type of the environment entry value, i.e. the type of the object returned from the JNDI `lookup` method.
- An optional `env-entry-value` element that specifies the value of the environment entry.
- An environment entry is scoped to the SBB component whose declaration contains the `env-entry` element. This means that the environment entry is inaccessible from other SBB components at runtime, and that other SBB components may define `env-entry` elements with the same `env-entry-name` without causing a name conflict.
- The environment entry values may be one of the following Java types: `String`, `Character`, `Integer`, `Boolean`, `Double`, `Byte`, `Short`, `Long`, and `Float`.
- If the SBB Developer provides a value for an environment entry using the `env-entry-value` element, the value can be changed later by the SBB Developer or Service Deployer. The value must be a string that is valid for the constructor of the specified type that takes a single `String` parameter, or for `java.lang.Character`, a single character.

The following example is the declaration of environment entries used by the Foo SBB whose code was illustrated in the previous subsection.

```
<sbb>
        …
        <sbb-classes>
                …
                <sbb-abstract-class>
                        …
                        <sbb-abstract-class-name>
                                com.foobar.FooSbb
                        </sbb-abstract-class-name>
                </sbb-abstract-class>
                …
        </sbb-classes>
        …
        <env-entry>
                <description>
                        The maximum number of connections allowed.
                </description>
                <env-entry-name> maxConnections </env-entry-name>
                <env-entry-type> java.lang.Integer </env-entry-type>
                <env-entry-value> 5 </env-entry-value>
        </env-entry>
        <env-entry>
                <description>
                        The minimum number of connections allowed.
                </description>
                <env-entry-name> minConnections </env-entry-name>
                <env-entry-type> java.lang.Integer </env-entry-type>
                <env-entry-value> 1 </env-entry-value>
```

```
        </env-entry>
        <env-entry>
                <env-entry-name> foo/name1 </env-entry-name>
                <env-entry-type> java.lang.String </env-entry-type>
                <env-entry-value> value1 </env-entry-value>
        </env-entry>
        <env-entry>
                <env-entry-name> foo/bar/name2 </env-entry-name>
                <env-entry-type> java.lang.Boolean </env-entry-type>
                <env-entry-value> true </env-entry-value>
        </env-entry>
        <env-entry>
                <description> Some description. </description>
                <env-entry-name> name3 </env-entry-name>
                <env-entry-type>java.lang.Integer </env-entry-type>
        </env-entry>
        <env-entry>
                <env-entry-name> foo/name4 </env-entry-name>
                <env-entry-type> java.lang.Integer </env-entry-type>
                <env-entry-value> 10 </env-entry-value>
        </env-entry>
        …
</sbb>
```

6.13.1.2 The Service Deployer's responsibility

The Service Deployer must ensure that the values of all the environment entries declared by the SBB component are set to meaningful values. The Service Deployer can modify the values of the environment entries that have been previously set by the SBB Developer, and must set the values of those environment entries for which no value has been specified.

The `description` elements provided by the SBB Developer help the Service Deployer with this task.

6.13.2 Resource adaptor type defined Activity Context Interface Factory object references

This section describes the programming interfaces and deployment descriptor elements that allow the SBB code to refer to resource adaptor type defined Activity Context Interface Factory objects using logical names called *resource adaptor type bindings*. These bindings are special entries in SBB component environments. The Service Deployer uses these bindings to bind actual resource adaptor type defined Activity Context Interface Factory objects at runtime to SBB components.

6.13.2.1 SBB Developer's responsibilities

This section describes the SBB Developer's view of locating resource adaptor type defined Activity Context Interface Factory objects and defines the accompanying responsibilities.

6.13.2.1.1 Programming interfaces for resource adaptor type bindings

The SBB Developer must use resource adaptor type bindings to obtain references to resource adaptor type defined Activity Context Interface Factory objects (see Section 7.6.1) as follows.

- Assign an entry in the SBB's component environment to the resource adaptor type binding. (See Section 6.13.2.1.2 for information on how resource adaptor type bindings are declared in the SBB deployment descriptor.)

  The SLEE specification recommends, but does not require, that all resource adaptor type bindings be organized in the subcontexts of the SBB's component environment, using a different subcontext for each resource adaptor type under the `java:comp/env/slee/resources` subcontext*. For example, all JCC resource adaptor type bindings might be declared in the* `java:comp/env/slee/resources/jcc` *subcontext, and all JAIN SIP resource adaptor type bindings in the* `java:comp/env/slee/resources/sip` *subcontext.*

- Lookup the resource adaptor type defined Activity Context Interface Factory objects in the SBB's component environment using the JNDI interface.

The following code sample illustrates obtaining a JCC Activity Context Interface Factory object.

```
{
        …
        JccCall call = …
        …
        JccActivityContextInterfaceFactory acif =
                (JccActivityContextInterfaceFactory) new InitialContext().lookup
                ("java:comp/env/slee/resources/jcc/1.1/activitycontextinterfacefactory");
        ActivityContextInterface aci = acif.getActivityContextInterface(call);
        …
}
```

6.13.2.1.2 Declaration of resource adaptor type bindings in the SBB deployment descriptor

Although a resource adaptor type binding is an entry in the SBB's component environment, the SBB Developer must not use an `env-entry` element to declare it.

Instead, the SBB Developer must declare all the resource adaptor type bindings in the SBB deployment descriptor using `resource-adaptor-type-binding`. This allows the SBB Developer and Service Deployer to discover all the resource adaptor type bindings used by an SBB component.

Each `resource-adaptor-type-binding` element describes a single resource adaptor type binding. The `resource-adaptor-type-binding` element contains the following sub-elements:

- An optional `description` element.
- A `resource-adaptor-type-ref` element identifies the resource adaptor type of the resource adaptor type binding. It contains a `resource-adaptor-type-name` sub-element, a `resource-adaptor-type-vendor` sub-element, and a `resource-adaptor-type-version` sub-element. These sub-elements must match exactly the corresponding sub-elements in the `resource-adaptor-type` element (see Section 15.3.2) of the resource adaptor type. The `activity-context-interface-factory-interface-name` element of `resource-adaptor-type` element specifies the Java type of the Activity Context Interface Factory object returned from the JNDI `lookup` method.
- An `activity-context-interface-factory-name` element contains the name of the environment entry used in the SBB component's code to access an Activity Context Interface Factory object of the resource adaptor type. The name of the environment entry is relative to the `java:comp/env` context (e.g. the name should be `slee/resources/jcc/…` rather than `java:comp/env/slee/resources/jcc/…`).
- Zero or more resource-adaptor-entity-binding elements (see Section 6.13.3.1.2),

Each SBB component has its own set of resource adaptor type binding entries. The resource adaptor type binding environment entries are not shared with other SBB components. The SBB component environment may not be modified at runtime.

The following example is the declaration of resource adaptor type binding used by the example illustrated above.

```
<sbb>
        …
        <resource-adaptor-type-binding>
                <resource-adaptor-type-ref>
                        <resource-adaptor-type-name>
                                JCC
                        </resource-adaptor-type-name>
                        <resource-adaptor-type-vendor>
                                javax.csapi.cc.jcc
                        </resource-adaptor-type-vendor>
```

```
                        <resource-adaptor-type-version>
                                1.1
                        </resource-adaptor-type-version>
                </resource-adaptor-type-ref>
                <activity-context-interface-factory-name>
                        slee/resource/jcc/1.1/activitycontextinterfacefactory
                </activity-context-interface-factory-name>
                …
        </resource-adaptor-type-binding>
…
</sbb>
```

6.13.2.2 The Service Deployer's responsibility

The Service Deployer must ensure that the `resource-adaptor-type-ref` elements of all the `resource-adaptor-type-binding` elements declared by the SBB component are set to meaningful values. For example, a `resource-adaptor-type-ref` must refer to a resource adaptor type that is already installed in the SLEE. Otherwise, the SBB cannot be installed in the SLEE. The Service Deployer must set the values of those `resource-adaptor-type-ref` elements for which no value has been specified.

The `description` elements provided by the SBB Developer help the Service Deployer with this task.

6.13.3 Resource adaptor interface object references

Changed in 1.1: Revised to align with the Resource Adaptor changes in 1.1. Resource adaptor objects in 1.0 changed to resource adaptor interface objects in 1.1.

This section describes the programming interfaces and deployment descriptor elements that allow the SBB code to refer to resource adaptor interface objects using logical names called *resource adaptor entity bindings*. A resource adaptor interface object is an object that implements the resource adaptor interface of a resource adaptor type. The resource adaptor entity bindings are special entries in SBB component environments. The Service Deployer uses these bindings to bind actual resource adaptor interface objects provided by resource adaptor entities at runtime to SBB components.

6.13.3.1 SBB Developer's responsibilities

This section describes the SBB Developer's view of locating resource adaptor interface objects and defines the accompanying responsibilities.

6.13.3.1.1 Programming interfaces for resource adaptor entity bindings

The SBB Developer must use resource adaptor entity bindings to obtain references to resource adaptor interface objects as follows.

- Assign an entry in the SBB's component environment to the resource adaptor entity binding. (See Section 6.13.3.1.2 for information on how resource adaptor entity bindings are declared in the SBB deployment descriptor.)

  The SLEE specification recommends, but does not require, that all resource adaptor entity bindings be organized in the subcontexts of the SBB's component environment, using a different subcontext for each resource adaptor entity binding under the `java:comp/env/slee/resources` *subcontext. For example, all resource adaptor entity bindings of the JCC resource adaptor type might be declared in the* `java:comp/env/slee/resources/jcc` *subcontext, and all JAIN SIP resource adaptor entity bindings in the* `java:comp/env/slee/resources/sip` *subcontext.*

- Lookup the resource adaptor interface objects in the SBB's component environment using the JNDI interface.

The following code sample illustrates obtaining a `JccProvider` object. The `JccProvider` interface is the resource adaptor interface of the JCC resource adaptor type.

```
{
        …
        JccProvider provider = (JccProvider) new InitialContext().lookup
                ("java:comp/env/slee/resources/jcc/1.1/jcc_1");
        JccCall call = provider.createCall();
        …
}
```

6.13.3.1.2 Declaration of resource adaptor entity bindings in the SBB deployment descriptor

Although a resource adaptor entity binding is an entry in the SBB's component environment, the SBB Developer must not use an `env-entry` element to declare it.

Instead, the SBB Developer must declare all the resource adaptor entity bindings in the SBB deployment descriptor using `resource-adaptor-entity-binding`. This allows the SBB Developer and Service Deployer to discover all the resource adaptor entity bindings used by an SBB component.

Each `resource-adaptor-entity-binding` element describes a single resource adaptor entity binding and is enclosed within a `resource-adaptor-type-binding` element. The `resource-adaptor-entity-binding` element contains the following sub-elements:

- An optional `description` element.
- A `resource-adaptor-object-name` element contains the name of the environment entry used in the SBB component's code to access the target resource adaptor interface object of the resource adaptor entity binding. The name of the environment entry is relative to the `java:comp/env` context (e.g., the name should be `slee/resources/jcc/…` rather than `java:comp/env/slee/resources/jcc/…`).
- A `resource-adaptor-entity-link` element identifies the resource adaptor entity that will provide the resource adaptor interface object that should be bound into the JNDI component environment of the SBB. The name specified in the `resource-adaptor-entity-link` element must be equal to a link name that a resource adaptor entity has been bound to using the `bindLinkName` method of the `ResourceManagementMBean` interface (refer Section 14.12). The resource adaptor entity bound to the specified link name must be an instance of a resource adaptor that implements the resource adaptor type specified by the `resource-adaptor-type-ref` sub-element of the enclosing `resource-adaptor-type-binding` element.

Each SBB component has its own set of resource adaptor entity binding entries. All instances of classes of an SBB component share the same resource adaptor entity binding environment entries; the resource adaptor entity binding environment entries are not shared with other SBB components. Instances of the classes of the SBB component are not allowed to modify the SBB component environment at runtime.

The following example is the declaration of resource adaptor entity binding used by the example illustrated above.

```
<sbb>
        …
        <resource-adaptor-type-binding>
                …
                <resource-adaptor-type-ref>
                        <resource-adaptor-type-name>
                                JCC
                        </resource-adaptor-type-name>
                        <resource-adaptor-type-vendor>
                                javax.csapi.cc.jcc
                        </resource-adaptor-type-vendor>
                        <resource-adaptor-type-version>
```

```
                                1.1
                        </resource-adaptor-type-version>
                </resource-adaptor-type-ref>
                …
                <resource-adaptor-entity-binding>
                        <resource-adaptor-object-name>
                                slee/resources/jcc/1.1/jcc_1
                        </resource-adaptor-object-name>
                        <resource-adaptor-entity-link>
                                …
                        </resource-adaptor-entity-link>
                </resource-adaptor-entity-binding>
        …
        </resource-adaptor-type-binding>
</sbb>
```

6.13.3.2 The Service Deployer's responsibility

Changed in 1.1: Allow Services to be deployed without "resolving" `resource-adaptor-entity-link` *elements. These links must be satisfied prior to activation of a Service rather than at deployment time.*

The Service Deployer must ensure that the `resource-adaptor-entity-link` elements of all the `resource-adaptor-entity-binding` elements declared by the SBB component are set to meaningful values. Prior to activating a Service each `resource-adaptor-entity-link` element specified by the SBBs within the Service must refer to a resource adaptor entity that is already instantiated in the SLEE. Otherwise, the Service cannot be activated in the SLEE. The Service Deployer must set the values of those `resource-adaptor-entity-link` elements for which no value has been specified.

The `description` elements provided by the SBB Developer help the Service Deployer with this task.

6.13.4 EJB references

Clarified from 1.1: The 1.0 specification references Section 20.3 in "Enterprise JavaBeans 2.0, Final Release" for a description of the programming interfaces and deployment descriptor elements that allow the SBB code to reference EJBs. The 1.1 specification includes a standalone description for EJB references.

This section describes the programming and deployment descriptor interfaces that allow the SBB to refer to the homes of EJBs using "logical" names called EJB references. The EJB references are special entries in the SBB's environment. The Service Deployer binds the EJB references to the EJB homes in the target deployment environment.

6.13.4.1 SBB Developer's responsibilities

This subsection describes the SBB Developer's view and responsibilities with respect to EJB references.

6.13.4.1.1 EJB reference programming interfaces

The SBB must use EJB references to locate the home interfaces of EJBs as follows.

- Assign an entry in the SBB's environment to the reference. (See Section 6.13.4.1.2 for information on how EJB references are declared in the deployment descriptor.)
  The SLEE specification recommends, but does not require, that all references to other EJBs be organized in the `ejb` *subcontext of the SBB's environment (i.e., in the* `java:comp/env/ejb` *JNDI context).*
- Look up the EJB home interface of the referenced EJB in the SBB's environment using JNDI.

The following example illustrates how an SBB uses an EJB reference to locate the remote home interface of an EJB.

```
{
        …
        Object result = new InitialContext().lookup
```

```
                ("java:comp/env/ejb/EmplRecord");
        EmployeeRecordHome emplRecordHome = (EmployeeRecordHome)
                javax.rmi.PortableRemoteObject.narrow(result,EmployeeRecordHome.class);
        …
}
```

In the example, the SBB Developer of the SBB assigned the environment entry `ejb/EmplRecord` as the EJB reference name to refer to the remote home of referenced EJB.

6.13.4.1.2 Declaration of EJB references in deployment descriptor

Changed in 1.1: The `ejb-link` element has been removed. The method in which an EJB of the appropriate type is identified to the SLEE is SLEE-vendor specific.

Although the EJB reference is an entry in the SBB's environment, the SBB Developer must not use a `env-entry` element to declare it. Instead, the SBB Developer must declare all the EJB references using the `ejb-ref` element of the deployment descriptor. This allows the SBB Developer and Service Deployer to discover all the EJB references used by an SBB component.

Each `ejb-ref` element describes the interface requirements that the referencing SBB has for the referenced EJB. The `ejb-ref` element is used for referencing an EJB that is accessed through its remote home and remote interfaces.

The `ejb-ref` element contains the following sub-elements:

- An optional `description` element.
- An `ejb-ref-name` element specifies the EJB reference name; its value is the environment entry name used in the SBB code relative to the `java:comp/env` context.
- A `ejb-ref-type` element specifies the expected type of the EJB; its value must be either Entity or Session.
- A `home` element specifies the expected Java type of the referenced EJB's remote home interface.
- A `remote` element specifies the expected Java type of the referenced EJB's remote component interface.

Each SBB component has its own set of EJB references. The EJB reference environment entries are not shared with other SBB components. The SBB component environment may not be modified at runtime.

The following example is the declaration of EJB references used by the example illustrated above.

```
<sbb>
        …
        <ejb-ref>
                <description>
                        This is a reference to the entity bean that
                        encapsulates access to employee records.
                </description>
                <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
                <ejb-ref-type>Entity</ejb-ref-type>
                <home>com.wombat.empl.EmployeeRecordHome</home>
                <remote>com.wombat.empl.EmployeeRecord</remote>
        </ejb-ref>
        …
</sbb>
```

6.13.4.2 The Service Deployer's responsibility

The Service Deployer must use the tools provided by the SLEE vendor to identify the objects that will be bound into an SBB's component environment for each of the `ejb-ref` elements declared by the SBB

component[6]. Otherwise, the SBB cannot be installed in the SLEE. The `description` elements provided by the SBB Developer help the Service Deployer with this task.

The Service Deployer must ensure that the target EJB is type-compatible with the declared EJB reference. This means that the target EJB must be of the type indicated in the `ejb-ref-type` element, and that the `home` and `remote` interfaces of the target EJB must be Java type-compatible with the interfaces declared in the EJB reference.

6.13.5 The SLEE's responsibility

The SLEE has the following responsibilities:

- Implement the `java:comp/env` environment naming context, and provide it to SBB Objects at runtime. The naming context must include all the environment entries, resource adaptor type bindings, resource adaptor entity bindings, and EJB references declared by the SBB Developer with their values supplied (directly or indirectly) by their respective deployment descriptor elements. The environment naming context must allow creation of subcontexts if they are needed by an SBB.
- The SLEE must ensure that the instances of the SBB component classes have only read access to their component environment. The SLEE must throw the `javax.naming.OperationNotSupportedException` from all the methods of the `javax.naming.Context` interface that modify the environment naming context and its subcontexts.

6.14 Operations allowed in the methods of the SBB abstract class

Method	***Method can perform the following operations***	***Transaction***
Constructor	-	Unspecified

`setSbbContext`
`unsetSbbContext`
`sbbActivate`
`sbbPassivate`
initial event selector methods
	JNDI access to `java:comp/env`	Unspecified

`sbbCreate`	JNDI access to `java:comp/env` Resource access CMP field access Activity Context attribute access: cannot attach or detach Fire events using fire event methods	Mandatory

`sbbPostCreate`
`sbbRemove`
`sbbLoad`
`sbbStore`
event handler method
local interface method
`sbbRolledBack`
	JNDI access to `java:comp/env`
	Resource access
	CMP field access
	Activity Context access: includes attributes, attach, and detach
	Event mask access
	Fire events using fire event methods
	Create child SBB entities
	Invoke methods on SBB local objects
		Mandatory

`sbbExceptionThrown`	Same as the method invoked by the SLEE that returned by throwing a `RuntimeException`. *While changes to transactional state, like creating child SBB entities, changes to CMP fields, Activity Contexts, etc, are allowed if the SBB object invoked is in the Ready state, these changes will be rolled back on*	Mandatory

[6] For example this may be done by providing a vendor-specific supplementary deployment descriptor.

	return from an invocation of the `sbbException-Thrown` *method.*	

6.15 SBB abstract class methods invoked by the SBB abstract class

Abstract class or interface	***Method***	***Transaction***
SBB abstract class	fire event methods get child relation methods CMP field accessor methods `asSbbActivityContextInterface` get Profile CMP methods	Mandatory

`SbbContext` interface
(Clarified in 1.1)
	`getService`
	`getSbb`
	`getTracer`
		Non-transactional

	`getSbbLocalObject` `getActivities`	Mandatory
	`maskEvent` `getEventMask`	Mandatory
	`setRollbackOnly` `getRollbackOnly`	Mandatory
`ChildRelation` interface	all methods	Mandatory

SBB local interface	`remove`
	`isIdentical`
	`getSbbPriority`
	`setSbbPriority`
	SBB specific local interface methods
		Mandatory

Chapter 7 Activity Objects, Activity Contexts, and Activity Context Interface Objects

The SLEE specification specifies the interface of the Java objects used to interact with Activities and Activity Contexts as well as the relations among these Java objects, Activities, and other SLEE specification defined entities.

- Activity.
  An Activity represents a related stream of one or more events. These events represent occurrences of significance that have occurred on the entity represented by the Activity. From a resource's perspective, an Activity represents a state machine within the resource that emits messages or events on state changes within the resource.
- Activity object.
  An Activity object is a Java representation of an Activity. It is an object that provides an Activity with a visible Java API that can be invoked. Activity objects are defined, created and owned by resource adaptor entities or SLEE Facilities.
- Activity Context.
  The SLEE uses an Activity Context to represent and encapsulate an underlying Activity object within the SLEE. An Activity Context holds shareable attributes that SBB entities use to share state relating to an Activity object and is a logical channel that both emits events and accepts events fired on the channel. An Activity Context is a logical entity within the SLEE. It does not have a visible Java API.
- Activity Context Interface object.
  An SBB object interacts with an Activity Context through an Activity Context Interface object. The Activity Context Interface object provides an Activity Context with a visible Java API that can be invoked. All Activity Context Interface objects implement the generic `ActivityContextInterface` interface and may optionally implement an extended SBB Activity Context Interface interface (see below). The generic `ActivityContextInterface` interface does not declare any shareable attributes. There are two kinds of Activity Context Interface objects.
  - Generic Activity Context Interface object.
    A generic Activity Context Interface object implements the generic `ActivityContextInterface` interface but does not implement an extended SBB Activity Context Interface interface. It cannot be used to access attributes stored in an Activity Context.
  - SBB Activity Context Interface object.
    An SBB may define an SBB Activity Context Interface interface that extends the generic `ActivityContextInterface` interface. An SBB Activity Context Interface object implements an SBB Activity Context Interface interface. The SBB Activity Context Interface interface declares the shareable attributes of the SBB (see Section 7.4). The SBB Activity Context Interface interface provides a type-safe mechanism for accessing shareable attributes.

7.1 Relations

The SLEE architecture defines the following relations:

- Activity objects and Activity Contexts: 1-to-1.
- Activity Contexts and Activity Context Interface objects: 1-to-many.
- Activity Contexts and SBB entities: many-to-many.
- Activity Contexts and SLEE Facilities: many-to-many

7.1.1 Activity objects and Activity Contexts

Changed in 1.1: This section includes the Activity Handle.

The following illustration shows the one-to-one relationship between an Activity object and an Activity Context. Typically, an Activity object resides within the resource domain. An Activity Context resides in the SLEE domain to represent an underlying Activity object in the resource domain.

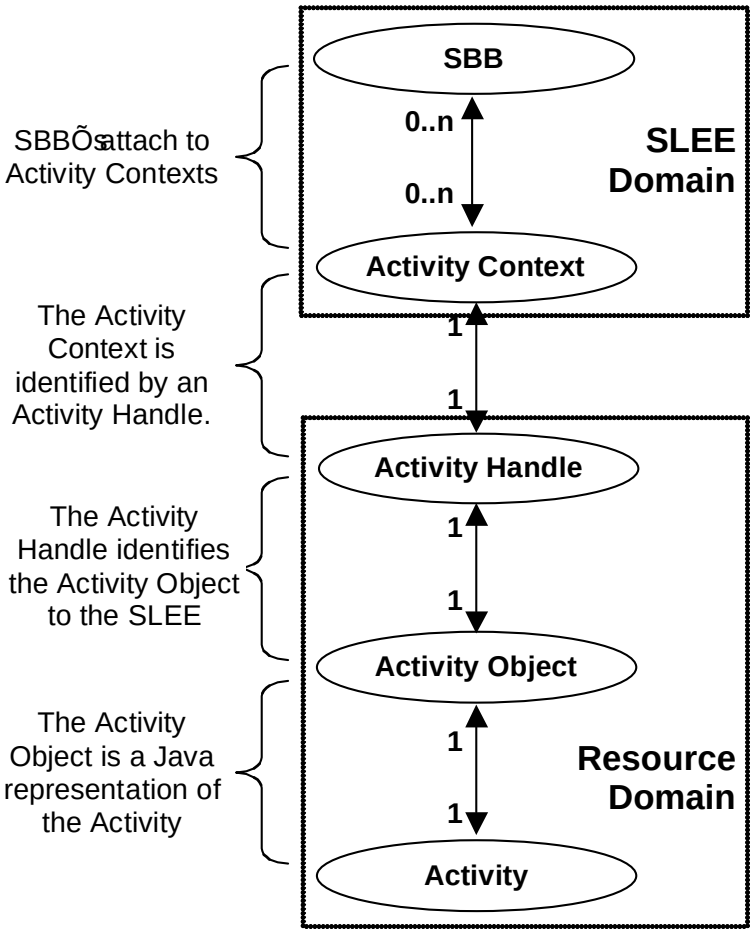

Figure 6 Activity objects and Activity Contexts

7.1.2 Activity Contexts and Activity Context Interface objects

The following illustration shows the one-to-many relationship between an Activity Context and Activity Context Interface objects.

Logically, an Activity Context has a generic Activity Context Interface object and an SBB Activity Context Interface object for each SBB that defines an SBB Activity Context Interface interface. Each of these SBB Activity Context Interface objects implements a different SBB Activity Context Interface interface.

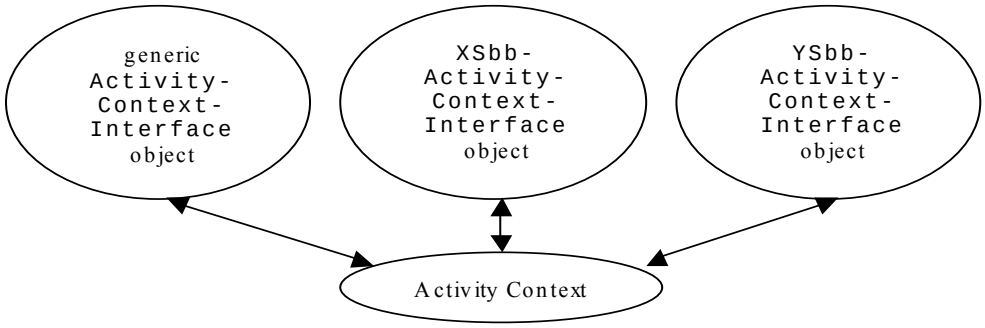

Figure 7 An Activity Context and its Activity Context Interface objects

7.1.3 Activity Contexts and SBB entities

There is a many-to-many relation between Activity Contexts and SBB entities. SBB entities are *attached* to Activity Contexts. The SLEE specification uses the term *attachment* to refer to this relation.

- Zero or more SBB entities can attach to an Activity Context.
- Each SBB entity can be attached to zero or more Activity Contexts.
- An SBB entity can be attached to a particular Activity Context zero or one times. *(Clarified in 1.1)*
- An SBB entity will only receive events fired on Activity Contexts that it has been attached to. The SLEE automatically attaches an SBB entity to an Activity Context when the SBB entity receives an initial event delivered on the Activity Context (see Section 8.6). An SBB entity may also explicitly attach itself to an Activity Context.
- The SLEE automatically detaches an SBB entity from an Activity Context after the SLEE delivers an Activity End Event to the SBB entity. The SLEE also automatically detaches an SBB entity from all Activity Contexts before the SBB entity is removed. Alternatively, an SBB entity may also be explicitly detached from an Activity Context.

7.1.4 Activity Contexts and SLEE Facilities

There is a many-to-many relation between Activity Contexts and SLEE Facilities. The SLEE specification uses *reference* to refer to this relationship.

- Each SLEE Facility can reference zero or more Activity Contexts. Examples of SLEE Facilities that maintain references to Activity Contexts include the Timer Facility and the Activity Context Naming Facility.
- Each Activity Context can be *referenced by* zero or more SLEE Facilities.

7.2 Activity object

An Activity object has a Java type of `java.lang.Object` in the SLEE API. The SLEE specification does not define a more specific Java type for every Activity object. The SLEE and resource adaptors can define more specific types for Activity objects. These types are always `Object`s. For example, the JCC resource adaptor type specified `JccCall` and `JccConnection` Activity objects. The SLEE and Resource Adaptor entities can create new Activities.

7.2.1 Sources of Activity objects

Main sources of Activity objects include:

- The SLEE.
  The SLEE defines the following Activity objects.

  - `NullActivity` – A `NullActivity` object represents an Activity that has no external source of events (see Section 7.10).
  - `ServiceActivity` - The SLEE fires Service lifecycle Events on `ServiceActivity` objects (see Section 8.8)
  - `ProfileTableActivity` - The SLEE fires Profile Table and Profile lifecycle events on `ProfileTableActivity` objects (see Section 13.6).

- Resource adaptor entities.
  A resource adaptor type declares the Java type of Activity objects common to a set of resource adaptors. Logically, the resource adaptor entities instantiated from these resource adaptors create these Activity objects and control the lifetime of these Activity objects. The Java types of these Activity objects may be defined by the underlying resources represented by the resource adaptor type. The Java types of Activity objects defined by the underlying resource may include, for example:
  - `EmergencyReport` and `CamelReport` from User Location and Status.
  - `JccConnection` and `JccCall` from Java Call Control.
  - `ServerTransaction` and `ClientTransaction` from JAIN SIP.

  An underlying resource may have a notion of an Activity, but does not use a Java object to represent this Activity. In this case, the resource adaptor entity that represents the underlying resource must provide Java objects to represent these Activities. The resource adaptor type of the resource adaptor entity defines the Java types of these Java objects. For example, JAIN TCAP uses a Dialog ID to represent an Activity. The Java type of a Dialog ID is the integer primitive Java type. The JAIN TCAP resource adaptor type defines `java.lang.Integer` as the Java type of the Activity objects that represent Dialog ID Activities.

The source of an Activity object is also the owner of the Activity object since it controls the lifetime of the Activity object.

7.2.2 Getting an Activity object

There are many ways to obtain a reference to an Activity object. The more common ways include:

- Getting the Activity object from an Activity Context Interface object.
  An event handler method is always invoked with an event object and an Activity Context Interface object. The generic `ActivityContextInterface` interface implemented by all Activity Context Interface objects defines a method that returns an Activity Context Interface object's underlying Activity object (see Section 7.4).
- Getting the Activity object from an event object.
  An event object defines zero or more methods that return Activity objects. For example, a `JccConnectionEvent` object (from Java Call Control) has a `getCall` method to get the `JccCall` Activity object and a `getConnection` method to get the `JccConnection` Activity object associated with the `JccConnectionEvent` object.
- Invoking a resource adaptor entity provided object to create a new Activity object.
  For example, `JccCall` (from Java Call Control) defines two methods that create new `JccConnection` objects: `createConnection` and `routeCall`.
- Invoking a `NullActivityFactory` object to create a new `NullActivity` object (see Section 7.10.2).
- Invoking a `ServiceActivityFactory` object to obtain a `ServiceActivity` object (see Section 8.8.4).
- Invoking a `ProfileFacility` object to obtain a `ProfileTableActivity` object (see Section 13.6).

7.2.3 Firing events on an Activity object

The SLEE specification does not define any application visible API for firing an event on an Activity object. The SLEE specification does not define any fire event method that takes an Activity object as an argument.

7.2.4 Firing events on an Activity Context

The SLEE specification defines an API that SBBs use to fire events on an Activity Context (see Section 6.1.2).

7.2.5 Firing events on an Activity Handle

Added in 1.1.

The SLEE specification defines an API that Resource Adaptors use to fire events on an Activity Handle (see Section 15.14.8).

7.2.6 Every Activity object has an end

Every Activity object must have an end. The SLEE specification does not define how an Activity object ends. Logically, an Activity object ends when no additional events will be emitted from the Activity represented by the Activity object, except for the last Activity End Event. See Section 7.2.7 for the event type definition for the Activity End Event.

In most cases, the owner of the Activity object defines what causes the Activity object to end. For example, a `JccCall` Activity object may end implicitly when a network event occurs, like a party hanging up. Alternatively, application code may explicitly cause the phone call to end, like invoking the `release` method on the `JccCall` object that represents the phone call (see Section 7.3.4).

When an Activity object ends, the SLEE delivers an Activity End Event to all SBB entities that are attached to the Activity Context of the Activity object and can receive the Activity End Event (see Section 8.6.5).

7.2.7 Activity End Event

The definition of the Activity End Event is as follows. See Section 8.4.1 for how an event type is defined.

The `event-definition` element for Activity End Events is as follows:

```
<event-definition>
…
        <event-type-name> javax.slee.ActivityEndEvent </event-type-name>
        <event-type-vendor> javax.slee </event-type-vendor>
        <event-type-version> 1.0 </event-type-version>
        <event-class-name> javax.slee.ActivityEndEvent </event-class-name>
</event-definition>
```

The `ActivityEndEvent` interface is as follows:

```
package javax.slee;

public interface ActivityEndEvent {
}
```

7.3 Activity Context

An Activity Context is logical object within the SLEE. It has no visible API. SBB entities interact with Activity Contexts through Activity Context Interface objects (see Section 7.4). The SLEE uses an Activity Context to represent and encapsulate an underlying Activity object within the SLEE.

An Activity Context may also hold shareable attributes. Typically, an SBB entity stores information in the Activity Context to share this information with other SBB entities that are also manipulating the same underlying Activity object. SBB entities should not store per-instance data in Activity Contexts. Per-instance data should be stored in each SBB entity's CMP fields (see Section 6.5).

Any state stored in an Activity Context by an SBB entity is persistent for the lifetime of the Activity Context. An SBB entity detaching from an Activity Context can expect any information it has stored in the Activity Context to be persisted for later use by the same or another SBB entity.

7.3.1 Activity Context creation

Changed in 1.1: Activity Handle abstraction introduced.

Activity Contexts are created and owned by the SLEE. The SLEE creates a new Activity Context when it is presented with an Activity Handle for a new Activity object. Typically, this happens when:

- A resource adaptor entity indicates to the SLEE that an Activity, represented by the Activity Handle, is starting in the SLEE.
- An SBB entity invokes an Activity Context Interface Factory object to obtain a generic Activity Context Interface object for a new Activity object. (The Activity object may have been created by the SBB entity invoking a method on a resource object, or the Resource Adaptor may be lazily starting the activity in the SLEE as described in Section 15.12.5.3).

7.3.2 Activity Context operations

The logical operations that can be performed on an Activity Context include:

- Set and get Activity Context attributes.
  Collaborating SBB entities use these attributes to share state that are related to the underlying Activity object (see Section 7.4).
- Attach an SBB entity to an Activity Context (see Section 7.4.2) and detach an SBB entity from an Activity Context (see Section 7.4.3).
- Fire an event on an Activity Context (see Section 8.2.1).
- Deliver an event on an Activity Context (see Section 8.2.2).
- Set and get the event mask of an SBB entity attached to the Activity Context (see Section 8.5.3).

7.3.3 Activity Context state machine

The following diagram illustrates the Activity Context state machine. This state machine allows an Activity Context to “end gracefully” after the underlying Activity object has ended. It allows the SLEE to deliver events outstanding on the Activity Context. It also allows SBB entities to receive outstanding events after the underlying Activity object has ended.

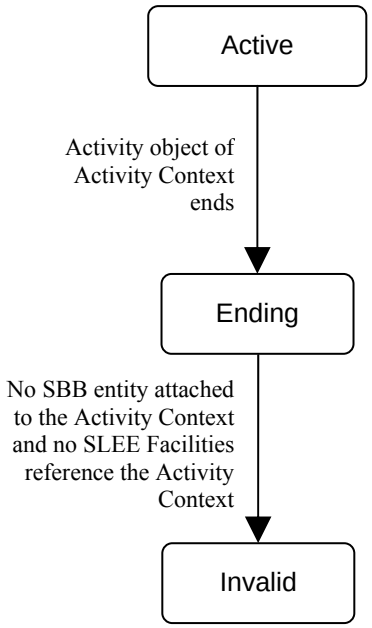

Figure 8 Life cycle of an Activity Context

An Activity Context can be in one of the following three states:

- Active.
  The Activity Context has been created. All the Activity Context operations are allowed.
- Ending.
  The Activity Context transitions to the Ending state from the Active state when the Activity object ends. All operations are allowed, except for firing more events on an Activity Context in the Ending state. If an SBB entity attempts to fire an event on an Activity Context in the Ending state, the SLEE will throw a `java.lang.IllegalStateException`. While in this state, the SLEE will continue to deliver outstanding events fired on the Activity Context. After the outstanding events have been delivered, the SLEE will fire an Activity End Event on the Activity Context. The SLEE delivers the Activity End Event to each interested SBB attached to the Activity Context and detaches each interested SBB entity from the Activity Context. The SLEE also detaches the SBB entities attached to the Activity Context that are not interested in the Activity End Event.
- Invalid
  The Activity Context transitions to the Invalid state from the Ending state when no SBB entity remains attached to the Activity Context (see Section 7.1.3) and no SLEE Facility references the Activity Context (see Section 7.1.4). No operation can be performed on an Activity Context in the Invalid state. An Activity Context in this state can be garbage collected or reclaimed by the SLEE.

The lifecycle state of an Activity Context is transactional (see Chapter 9). There is an implicit lifecycle state variable associated with each Activity Context that indicates the lifecycle state of the Activity Context. This lifecycle state variable can only be accessed or modified from within a transaction. For example, a fire event method reads this implicit lifecycle state variable whenever it is called. A fire event method must be invoked with a valid transaction. Alternatively, this lifecycle state variable can change asynchronously when an Activity Context ends asynchronously. This asynchronous change must occur within a transaction.

7.3.4 Activity and Activity Context end examples

This section presents a few examples on how an Activity object ends and how this alters the state of its associated Activity Context.

7.3.4.1 Explicit ending of a non-transactional Activity object example

Clarified in 1.1: Several conditions related to the ending of non-transactional activities have been clarified.

In this example, an SBB object invokes a method on an Activity object to end the underlying Activity. The Activity object is a hypothetical non-transactional `BazConnection` object owned by the BAZ resource adaptor entity and the end method is `close` (which is also non-transactional). Invoking this method immediately closes the underlying connection represented by the `BazConnection` object and no more events will be emitted by the underlying connection.

1. The SLEE starts a new transaction and invokes an event handler method of an SBB object within this transaction.
2. The event handler method invokes a `BazConnection` object's `close` method.
3. The BAZ resource adaptor entity closes the underlying connection.
4. The BAZ resource adaptor entity notifies the SLEE that the `BazConnection` object has ended.
5. The SLEE marks the Activity Context of the `BazConnection` object as being in the Ending state. Once the Activity Context is in the ending state the SLEE does not allow additional events to be fired on the Activity Context.
6. The event handler method returns and the SLEE commits the transaction successfully.
7. *Regardless of whether or not the transaction commits, the non-transacted activity will end.* The SLEE continues to deliver any events outstanding on the Activity Context.
8. The last outstanding event on the Activity Context has been delivered.
9. The SLEE performs the two following two sequences concurrently or in an unspecified order:
    a. The SLEE fires an Activity End Event on the Activity Context. The SLEE routes the Activity End Event to each SBB entity that is attached to the Activity Context and is interested in the Activity End Event (see Section 8.6). After the SLEE delivers the Activity End Event to an interested SBB entity, the SLEE detaches the SBB entity from the Activity Context.
    b. The SLEE also detaches SBB entities that are attached to the Activity Context but are not interested in the Activity End Event.
10. The SLEE notifies the SLEE Facilities that have references to the Activity Context that the Activity End Event has been delivered on the Activity Context. The Activity Context Naming Facility will remove the Activity Context's name bindings. The Timer Facility will remove any timers that depend on the Activity Context. Both of these Facilities will release their references to the Activity Context.
11. The SLEE transitions the Activity Context to the Invalid state and the SLEE can reclaim the Activity Context.

7.3.4.2 Explicit ending of a transactional Activity object example

Clarified in 1.1: Several conditions related to the ending of transactional activities have been clarified.

In this example, an SBB object invokes a method on an Activity object to end the Activity. The Activity object is a hypothetical transactional `ZaxConnection` object owned by the ZAX resource adaptor entity and the end method is `close` (which is also transactional). Invoking this method immediately closes the underlying connection represented by the `ZaxConnection` object and no more events will be emitted by the underlying connection.

1. The SLEE starts a new transaction and invokes an event handler method of an SBB object with this transaction.
2. The event handler method invokes a `ZaxConnection` object's `close` method.

3. The ZAX resource adaptor entity enrolls the underlying connection in the transaction and marks the isolated state for the connection as closing. As part of the commit processing the committed state for the connection is modified to be closing.
4. The event handler method returns and the SLEE commits the current transaction successfully. The ZAX resource adaptor entity participates in the transaction commit protocol and knows the outcome of the commit protocol.

   If the transaction rolls back, then the ZAX resource adaptor entity would be notified and it will remove its mark on the underlying connection and will not modify the committed state of the connection.
5. The ZAX resource adaptor entity closes the underlying connection.
6. The ZAX resource adaptor entity notifies the SLEE that the `ZaxConnection` object has ended.
7. The SLEE marks the Activity Context of the `ZaxConnection` object as being in the Ending state.

The remaining steps are the same as steps 7 through 11 in the Section 7.3.4.1.

7.3.4.3 An event ends an Activity object example

In this case, an event fired by a resource adaptor entity ends the Activity object. In this example, a `JccCall` object that represents a call ends and an event is fired indicating that no more events will be fired for the call.

1. The call ends. This call ends either due to:
   a. An SBB object ending the call as in the following case:
      i. The SLEE starts a new transaction and invokes an event handler method of an SBB object with this transaction.
      ii. The event handler method invokes a `JccCall` object's `release` method.
      iii. The event handler method returns and the SLEE commits the current transaction successfully.

      As the JCC resource is not a transactional resource the call will be released whether or not the transaction commits or is rolled back.
   b. One of the parties on a two party call hangs up.
2. The JCC resource adaptor entity releases the underlying call.
3. The JCC resource adaptor entity emits a `JccCallEvent` object whose `getID` method returns `JccCallEvent.CALL_EVENT_TRANSMISSION_ENDED`.
4. The JCC resource adaptor entity notifies the SLEE that the `JccCall` object has ended.
5. The SLEE marks the Activity Context of the `JccCall` object as being in the Ending state.

The remaining steps are the same as steps 7 through 11 in the Section 7.3.4.1.

7.4 Generic `ActivityContextInterface` interface

Changed in 1.1: Added the `isAttached` method.

All Activity Context Interface objects implement the generic `ActivityContextInterface` interface. The `ActivityContextInterface` interface is:

```
package javax.slee;

public interface ActivityContextInterface {
       public Object getActivity()
              throws  TransactionRequiredLocalException,
                      SLEEException;
       public void attach(SbbLocalObject sbbLocalObject)
```

```
                throws  NullPointerException,
                        TransactionRequiredLocalException,
                        TransactionRolledbackLocalException,
                        SLEEException;
        public void detach(SbbLocalObject sbbLocalObject)
                throws  NullPointerException,
                        TransactionRequiredLocalException,
                        TransactionRolledbackLocalException,
                        SLEEException;
        public boolean isAttached(SbbLocalObject sbbLocalObject)
                throws  NullPointerException,
                        TransactionRequiredLocalException,
                        TransactionRolledbackLocalException,
                        SLEEException;
        public boolean isEnding()
                throws  TransactionRequiredLocalException,
                        SLEEException;
        public boolean equals(Object other);
        public int hashCode();
}
```

All the methods of the `ActivityContextInterface` interface are mandatory transactional methods (see Section 9.6.1).

7.4.1 `ActivityContextInterface` interface `getActivity` method

The `ActivityContextInterface` interface has a `getActivity` method. This method returns the Activity object of an Activity Context.

- It is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

7.4.2 `ActivityContextInterface` interface `attach` method

The `ActivityContextInterface` interface has an `attach` method. This method attaches the SBB entity represented by the SBB local object specified by the `sbbLocalObject` argument to an Activity Context so that the specified SBB entity can receive events fired on the Activity Context.

- The `sbbLocalObject` argument must represent a valid SBB entity when this method is invoked. Otherwise, this method marks the current transaction for rollback and throws a `javax.slee.TransactionRolledbackLocalException`.
- This method throws a `java.lang.NullPointerException` if the `sbbLocalObject` argument is `null`.
- This method does not change the values of any Activity Context attributes.
- This method attaches the SBB entity specified by the `sbbLocalObject` argument to the Activity Context. An SBB that is attached to an Activity Context is eligible to receive events fired on the Activity Context.
- This method also sets the event mask of the specified SBB entity for the Activity Context to the set of event types whose `mask-on-attach` attribute of their `event` elements within the `sbb` element of the specified SBB is true (see Section 8.4.2).
- Invoking this method for an SBB entity that is already attached to the Activity Context has no effect. The event mask of the specified SBB entity for the Activity Context and the delivered set for any event being routed on the Activity Context at the time does not change in this case. *(Clarified in 1.1)*

- It is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- It throws a `javax.slee.SLEEException` if the SBB entity cannot be attached due to a system-level failure.

See Section **B.2** for the pseudo code that illustrates the desired behavior of the `attach` method.

This method should not be used to attach an SBB entity to the Activity Context of the Activity Context Interface object passed to the SBB entity through an event handler method since the SBB entity must already be attached to this Activity Context, in order for the SBB entity to receive an event through the event handler method (see Section 8.6).

7.4.3 `ActivityContextInterface` interface `detach` method

The `ActivityContextInterface` interface has a `detach` method. This method detaches the SBB entity represented by the SBB local object specified by the `sbbLocalObject` argument from an Activity Context so that the specified SBB entity will no longer receive events fired on the Activity Context.

- The `sbbLocalObject` argument must represent a valid SBB entity when this method is invoked. Otherwise, this method marks the current transaction for rollback and throws a `javax.slee.TransactionRolledbackLocalException`.
- This method throws a `java.lang.NullPointerException` if the `sbbLocalObject` argument is `null`.
- This method does not change the values of any Activity Context attributes.
- This method detaches the SBB entity specified by the `sbbLocalObject` argument from the Activity Context.
- Invoking this method for an SBB entity that is already unattached from the Activity Context has no effect. *(Clarified in 1.1)*
- It is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- It throws a `javax.slee.SLEEException` if the SBB entity cannot be detached due to a system-level failure.

See Section B.3 for the pseudo code that illustrates the desired behavior of the `detach` method.

7.4.4 `ActivityContextInterface` interface `isAttached` method

Added in 1.1

The `ActivityContextInterface` interface has an `isAttached` method. This method returns whether or not the SBB local object specified by the `sbbLocalObject` argument is currently attached to the Activity Context

- The `sbbLocalObject` argument must represent a valid SBB entity when this method is invoked. Otherwise, this method marks the current transaction for rollback and throws a `javax.slee.TransactionRolledbackLocalException`.
- This method throws a `java.lang.NullPointerException` if the `sbbLocalObject` argument is `null`.
- This method returns `true` if the SBB entity specified by the `sbbLocalObject` argument is current attached to the Activity Context. If the SBB entity is not attached, this method returns `false`.
- It is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.

- It throws a `javax.slee.SLEEException` if the SBB entity cannot be detached due to a system-level failure.

7.4.5 `ActivityContextInterface` interface `isEnding` method

The `ActivityContextInterface` interface has an `isEnding` method. This method returns whether the Activity Context is in the Ending state.

- It is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

7.4.6 `ActivityContextInterface` interface `equals` method

Added in 1.1

The `ActivityContextInterface` interface has an `equals` method. This method returns true if the invoked `ActivityContextInterface` represents the same Activity as another `ActivityContextInterface`.

7.4.7 `ActivityContextInterface` interface `hashCode` method

Added in 1.1

The `ActivityContextInterface` interface has a `hashCode` method. If two `ActivityContextInterface` objects are equal (according to the `equals` method) then they must return the same hash code.

7.5 SBB Activity Context Interface interface

An SBB may define an SBB Activity Context Interface interface that extends, either directly or indirectly, the generic `ActivityContextInterface` interface to declare the shareable attributes of the SBB. Every shareable attribute has an attribute name. The SLEE generates the concrete class that implements the SBB Activity Context Interface interface when the SBB is deployed into the SLEE as a part of a deployable unit.

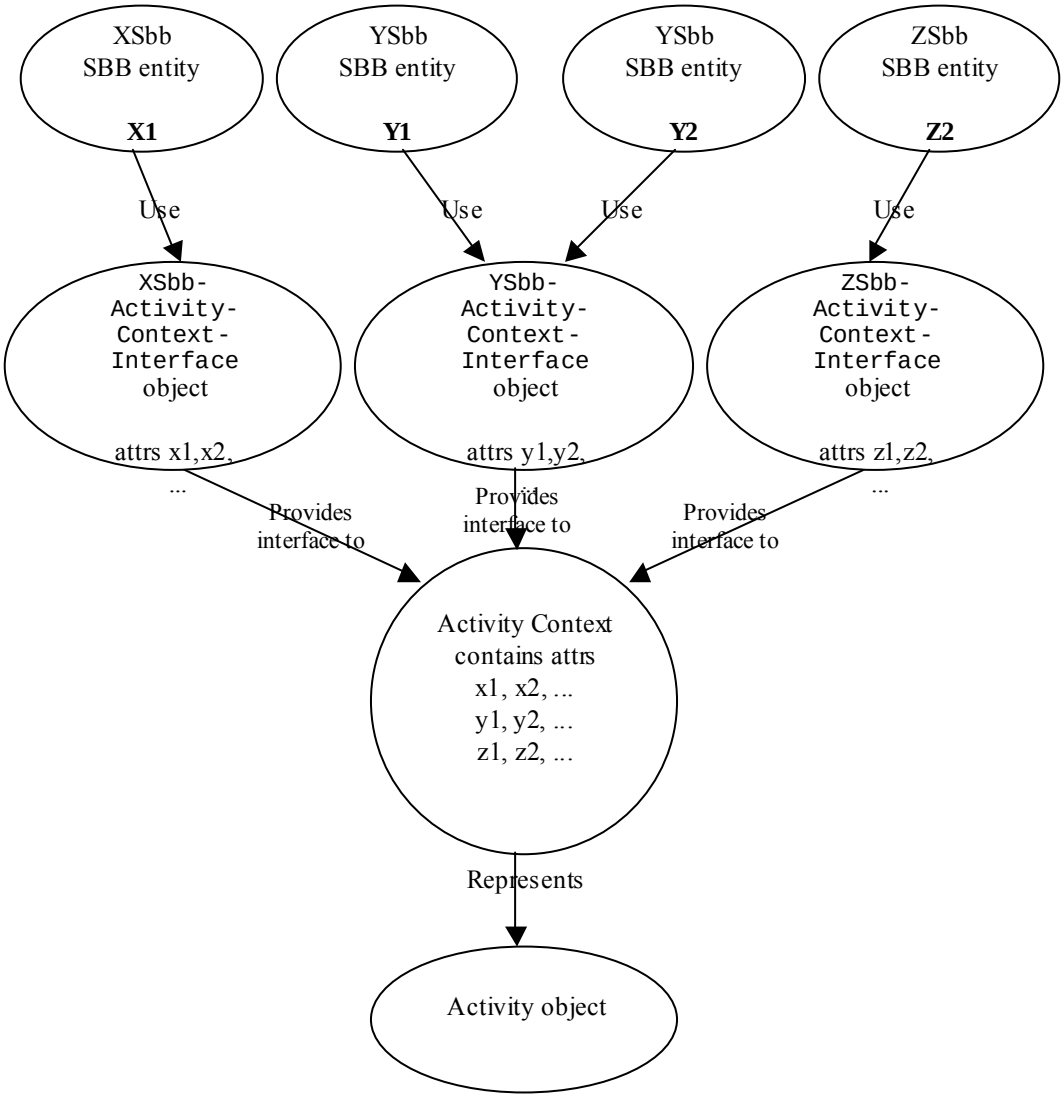

Figure 9 SBB Activity Context Interface objects example

If an SBB does not define an SBB-specific SBB Activity Context Interface interface, the SBB uses the generic `ActivityContextInterface` interface. Since the generic `ActivityContextInterface` interface does not declare any shareable attributes, this SBB will not be able to get or modify any attributes in any Activity Contexts.

- The SBB Activity Context Interface must be defined in a named package, i.e. the class must have a `package` declaration. *(Added in 1.1)*
- The SBB Activity Context Interface interface must be declared as `public`.

7.5.1 SBB Activity Context Interface get and set accessor methods

The SBB Developer must declare a set accessor method and/or a get accessor method for each shareable attribute in the SBB Activity Context Interface interface.

- The rules for computing accessor method names from an attribute name are the same as the rules for computing accessor method names from a CMP field name (see Section 6.5.1).
- The Java types of an Activity Context attribute are restricted to Java primitive types and Java serializable types. SBB local interfaces (i.e., `SbbLocalObject` and the types derived from `SbbLocalObject`) cannot be stored in Activity Context attributes.
- The semantics of these accessor methods are also the same as the semantics of CMP field accessor methods.
- For an attribute to be writeable, the SBB Developer must define a set accessor method. For an attribute to be readable, the SBB Developer must define a get accessor method. An attribute can be (from the SBB's point of view), either read-only, write-only or read-write, based on the methods defined.
- Activity Context attribute accessor methods are mandatory transactional methods (see Section 9.6.1).

In the following example, the `FooSbbActivityContextInterface` interface declares two attributes.

```
…
import javax.slee.ActivityContextInterface;
…
public interface FooSbbActivityContextInterface extends ActivityContextInterface {
        public int getFooAmount();
        public void setFooAmount(int fooAmount);
        public String getTestString();
        public void setTestString(String testString);
}
```

7.6 Getting an Activity Context Interface object

There are several ways for an SBB object executing on behalf of an SBB entity to obtain a reference to an Activity Context Interface object. The most common ways include:

- The SBB abstract class has an event handler method. When the SLEE delivers an event to the SBB entity, the SLEE invokes the event handler method, passing to the event handler method an event object and an Activity Context Interface object (see Section 8.5.2).
- The SBB object invokes an Activity Context Interface Factory object to get a generic Activity Context Interface object from an Activity object.
- The SBB retrieves a generic Activity Context Interface object from the Activity Context Naming Facility (see Section 7.9).

7.6.1 Activity Context Interface Factory objects

Every Activity object source has an Activity Context Interface Factory object. The Activity Context Interface Factory object has factory methods. Each factory method "manufactures" generic Activity Context Interface objects from Activity objects of a specific type defined by the Activity object source.

For example, the `JccActivityContextInterfaceFactory` interface for Java Call Control may look like this:

```
…
import javax.slee.ActivityContextInterface;
import javax.slee.FactoryException;
import javax.slee.UnrecognizedActivityException;
import javax.slee.TransactionRequiredLocalException;
…
```

```
public interface JccActivityContextInterfaceFactory {
       public ActivityContextInterface
              getActivityContextInterface(JccCall call)
                     throws  NullPointerException,
                             TransactionRequiredLocalException,
                             UnrecognizedActivityException,
                             FactoryException;
       public ActivityContextInterface
              getActivityContextInterface(JccConnection connection)
                     throws  NullPointerException,
                             TransactionRequiredLocalException,
                             UnrecognizedActivityException,
                             FactoryException;
}
```

- All factory methods are mandatory transactional methods.
- The SLEE specification defined `NullActivityContextInterfaceFactory` can be found in Section 7.10.
- The SLEE specification defined `ServiceActivityContextInterfaceFactory` can be found in Section 8.8.5.
- The SLEE specification defined `ProfileTableActivityContextInterfaceFactory` can be found in Section 10.9.5.
- The SLEE specification defined Activity Context Interface Factory objects are bound to well-defined locations in every SBB component environment.
- Activity Context Interface Factory interfaces defined by resource adaptor types must be defined in a named package, i.e. the interface must have a `package` declaration. *(Added in 1.1)*
- An SBB binds resource adaptor type defined Activity Context Interface Factory objects into its component environment using `resource-adaptor-type-binding` elements in the SBB's deployment descriptors (see Section 6.13.2).

A factory method may throw the following exceptions:

- `java.lang.NullPointerException`.
  A factory method throws this exception if its Activity object argument is `null`.
- `javax.slee.UnrecognizedActivityException`.
  A factory method throws this exception when the input argument does not represent an valid Activity for this Activity Context Interface Factory object, e.g. the Activity object is from a resource adaptor entity not served by this Activity Context Interface Factory object.
- `javax.slee.TransactionRequiredLocalException`.
  A factory method is a mandatory transactional methods (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- `javax.slee.FactoryException`.
  A factory method throws this exception if the method cannot create or return an `ActivityContextInterface` object because of system-level or SLEE-level problems. This exception is derived from `javax.slee.SLEEException`.

7.7 Activity Context methods in SBB abstract class and `SbbContext`

An SBB object uses the following methods provided in the `SbbContext` interface to interact with Activity Contexts.

- A `getActivities` method to get all the Activity Contexts attached to the SBB entity assigned to the SBB object.

- A `maskEvent` method specifies a set of event types that an SBB entity assigned to the SBB object does not want to receive from a specific Activity Context (see Section 8.5.3.1).
- A `getEventMask` method to obtain the set of event types that an SBB entity assigned to the SBB object does not want to receive from a specific Activity Context (see Section 8.5.3.2).

An SBB object uses the following method declared by the SBB Developer in the SBB abstract class to interact with Activity Contexts.

- An `asSbbActivityContextInterface` method to narrow an object that implements the generic `ActivityContextInterface` interface to an object that implements the SBB-specific SBB Activity Context Interface interface.

7.7.1 `SbbContext` interface `getActivities` method

The `SbbContext` object of every SBB object has a `getActivities` abstract method. An SBB object assigned to an SBB entity uses this method to get all the Activity Contexts that SBB entity is attached to.

The `getActivities` method looks like this:

```
public interface SbbContext {
        …
        public ActivityContextInterface[] getActivities()
                throws  TransactionRequiredLocalException,
                        IllegalStateException,
                        SLEEException;
        …
}
```

- This method does not change the values of any Activity Context attributes.
- The SBB object of the `SbbContext` object must have an assigned SBB entity when it invokes this method. Otherwise, this method throws a `java.lang.IllegalStateException`.
- It is a mandatory transactional method (see Section 9.6.1). Hence, the SBB object must invoke this method within a transaction. It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

See Section B.4 for the pseudo code that illustrates the desired behavior of the `getActivities` method.

7.7.2 SBB abstract class `asSbbActivityContextInterface` method

The SBB abstract class of an SBB that defines an SBB Activity Context Interface interface must declare an `asSbbActivityContextInterface` method.

This method takes as its input parameter an Activity Context Interface object and returns an object that implements the SBB Activity Context Interface interface of the SBB. The SBB Activity Context Interface interface provides the accessor methods that allow an SBB object to access the shareable attributes of the SBB that are stored in the Activity Context.

If the SBB does not define an SBB Activity Context Interface interface, then this method cannot be declared in the SBB abstract class.

The method signature of this method is as follows:

```
        public abstract <SBB Activity Context Interface>
                asSbbActivityContextInterface(ActivityContextInterface activity);
```

- The method must be declared as `public` and `abstract`. The SLEE provides the concrete implementation of the method when the SBB is deployed. *(Clarified in 1.1)*
- The return type must be the Activity Context Interface interface of the SBB, or a base interface of the Activity Context Interface interface of the SBB. *(Clarified in 1.1)*

- This method does not have `throws` clause.
- The SBB object must have an assigned SBB entity when it invokes this method. Otherwise, this method throws a `java.lang.IllegalStateException`.
- It is a mandatory transactional method (see Section 9.6.1). Hence, the SBB object must invoke this method within a transaction. It throws a `javax.slee.TransactionRequiredLocal-Exception` if it is invoked without a valid transaction context.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

7.8 Activity Context attribute aliasing

By default, the Activity Context attributes defined in the SBB Activity Context Interface interface of an SBB are not accessible by other SBBs. The attributes that an SBB entity stores in an Activity Context can only be accessed by SBB entities of the same SBB[7] because SBB entities of the same SBB use the same SBB Activity Context Interface object to interact with the Activity Context. This avoids unintentional shared access to the same attribute when composing SBBs, or when deploying Services from different sources.

Activity Context attribute aliasing directs the SLEE to make the attributes declared in different SBB Activity Context Interface interfaces behave logically as a single attribute. This logical attribute has a name known as the alias name. The logical attribute can be updated through any of its aliased attributes' set accessor methods. Changes to the logical attribute are observable through any of these attributes' get accessor methods. Shareable attributes defined in an SBB Activity Context Interface interface of an SBB are not "shared" with other SBBs unless explicitly aliased.

- The attributes that are to be aliased must be of the same Java type.
- An SBB Activity Context Interface attribute may be aliased at most once, i.e. it can either be unaliased or aliased through a single alias name.

7.8.1 Activity Context attribute aliasing deployment descriptor elements

The SBB Developer or Service Deployer specifies an Activity Context attribute alias using deployment descriptor elements in the `sbb` deployment descriptor element. Each Activity Context attribute alias is represented by an `activity-context-attribute-alias` element. An `activity-context-attribute-alias` element is a sub-element of the `sbb` element. See Section 3.1.8 for a description of the `activity-context-attribute-alias` element.

7.8.2 Activity Context attribute aliasing example

The following SBB deployment descriptor example aliases the `blockCounter` Activity Context attribute of the CallBlocking SBB and the `forwardCounter` Activity Context attribute of the CallForwarding SBB, and names the aliased logical attribute `counter`.

```
<sbb>
        …
        <sbb-name> CallBlockingSbb </sbb-name>
        <sbb-vendor> com.foobar </sbb-vendor>
        <sbb-version> 1.0 </sbb-version>
        …
        <activity-context-attribute-alias>
                <attribute-alias-name>
                        counter
                </attribute-alias-name>
                <sbb-activity-context-attribute-name>
```

[7] The SBB entities of the same SBB are the SBB entities that are instantiated from a single SBB component (which is installed in the SLEE only once) regardless of which Service the SBB entities are instantiated for.

```
                        blockCounter
                </sbb-activity-context-attribute-name>
        </activity-context-attribute-alias>
        …
</sbb>
…
<sbb>
        …
        <sbb-name> CallForwardingSbb </sbb-name>
        <sbb-vendor> com.foobar </sbb-vendor>
        <sbb-version> 1.0 </sbb-version>
        …
        <activity-context-attribute-alias>
                <attribute-alias-name>
                        counter
                </attribute-alias-name>
                <sbb-activity-context-attribute-name>
                        forwardCounter
                </sbb-activity-context-attribute-name>
        </activity-context-attribute-alias>
        …
</sbb>
```

7.9 Activity Context Naming Facility

A SLEE specification defined Activity Context Naming Facility provides a global flat namespace for naming Activity Contexts. It has methods to:

- Bind a name to an Activity Context.
  This operation causes the Activity Context Naming Facility to reference the Activity Context (see Section 7.1.4). A Null Activity can not implicitly transition from the Active to the Ending state, and can not be reclaimed while the Activity Context for the Null Activity is referenced by the Activity Context Naming Facility. *(Clarified in 1.1)*
- Find an Activity Context by name.
- Unbind an Activity Context entity from its name binding.
  This operation causes the Activity Context Naming Facility to not reference the Activity Context.

The Activity Context Naming Facility is used as follows:

- SBB objects use JNDI lookup to obtain an `ActivityContextNamingFacility` object from a SLEE specification defined location in every SBB component environment (see Section 6.13).
- The SBB Developers that use this Facility are responsible for choosing unique names to avoid unintentional naming conflicts.
- This Facility allows multiple SBB entities that have been created in response to initial events on different Activities (thus have different Activity Contexts) to converge on a "shared" Activity Context, so that these SBB entities can signal each other by firing events on the shared Activity Context and share common state by reading and modifying attributes.

7.9.1 Activity Context Naming Facility and Activity End Events

The Activity Context Naming Facility assists Activity Context reclamation by automatically unbinding an Activity Context from its name binding after the SLEE has delivered an Activity End Event to all SBB entities that are attached to the Activity Context and can receive the Activity End Event (see Section 8.6.5).

7.10 `NullActivity` objects

`NullActivity` objects do not process events from external resources. They are created and used by applications, and garbage collected by the SLEE. The only event which the SLEE fires on a `NullActivity` object is the ActivityEndEvent. SBBs may fire events on the Activity Context of a `NullActivity` object. *(Clarified in 1.1)*

Collaborating SBB entities can use the Activity Context of a `NullActivity` object as a private communications channel. They signal one another by firing events on the Activity Context of the `NullActivity` object and store shareable attributes in the Activity Context of the `NullActivity` object.

The SLEE makes a `NullActivityFactory` object accessible via an SBBs component environment. The `NullActivityFactory` object creates `NullActivity` objects. Hence, the SLEE is the owner of all `NullActivity` objects.

7.10.1 `NullActivity` interface

The SLEE specification defines `NullActivity` as a Java interface that contains a single `endActivity` method.

```
package javax.slee.nullactivity;

import javax.slee.SLEEException;
import javax.slee.TransactionRequiredLocalException;

public interface NullActivity {
       public void endActivity() throws TransactionRequiredLocalException, SLEEException;
}
```

- If the `NullActivity` object is already ending, subsequent `endActivity` method invocations are ignored.
- This method is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredException` if it is invoked without a valid transaction context.
- This method throws a `javax.slee.SLEEException` if the requested operation cannot be completed because of a system-level failure.

7.10.2 `NullActivityFactory` interface

Changed in 1.1: JNDI location constant added.

The SLEE specification defines the `NullActivityFactory` as a Java interface that contains a single `createNullActivity` method. The `NullActivityFactory` object is bound to a SLEE specification defined location in the component environment of every SBB (see Section 7.10.6).

```
package javax.slee.nullactivity;

import javax.slee.FactoryException;
import javax.slee.TransactionRequiredLocalException;

public interface NullActivityFactory {
       // JNDI location constant
       public static final String JNDI_NAME =
              "java:comp/env/slee/nullactivity/factory";

       public NullActivity createNullActivity()
              throws TransactionRequiredLocalException, FactoryException;
}
```

- The `JNDI_NAME` constant. *(Added in 1.1)*
  This constant specifies the JNDI location where a `NullActivityFactory` object may be located by an SBB component in its component environment.

- The `createNullActivity` method throws a `javax.slee.FactoryException` if the method cannot create or return a `NullActivity` object because of system-level or SLEE-level problems. This exception is derived from `javax.slee.SLEEException`.
- This method is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredException` if it is invoked without a valid transaction context.

7.10.3 `NullActivityContextInterfaceFactory` interface

Changed in 1.1: JNDI location constant added.

An SBB can obtain a generic Activity Context Interface object for a `NullActivity` object using a `NullActivityContextInterfaceFactory` object. The `NullActivityContextInterfaceFactory` object is also bound to a SLEE specification defined location in the component environment of every SBB.

```
package javax.slee.nullactivity;

import javax.slee.ActivityContextInterface;
import javax.slee.FactoryException;
import javax.slee.UnrecognizedActivityException;
import javax.slee.TransactionRequiredLocalException;

public interface NullActivityContextInterfaceFactory {
        // JNDI location constant
        public static final String JNDI_NAME =
                "java:comp/env/slee/nullactivity/activitycontextinterfacefactory";

        public ActivityContextInterface
                getActivityContextInterface(NullActivity activity)
                        throws  NullPointerException,
                                TransactionRequiredLocalException,
                                UnrecognizedActivityException,
                                FactoryException;
}
```

- The `JNDI_NAME` constant. *(Added in 1.1)*
  This constant specifies the JNDI location where a `NullActivityContextInterfaceFactory` object may be located by an SBB component in its component environment.
- See Section 7.6.1 for a general description on Activity Context Interface Factories, the transactional semantics, and the exceptions thrown by a `getActivityContextInterface` method.

7.10.4 All `NullActivity` methods are transactional

All the methods of the `NullActivity` interface, `NullActivityFactory` interface, and the `NullActivityContextInterfaceFactory` interface are mandatory transactional methods (see Section 9.6.1). For example, this means that a `NullActivity` object ends only if the transaction that invoked the `endActivity` method commits successfully.

7.10.5 Ending a `NullActivity` object

A `NullActivity` object may end either implicitly or explicitly. In both cases, the processing that occurs after the `NullActivity` object ends is the same as for all other Activity objects when they end. This processing is described in Section 7.3.3.

7.10.5.1 Implicitly ending a `NullActivity` object

Clarified in 1.1: Several conditions related to the implicit ending of Null Activity objects have been clarified.

The condition that implicitly ends a `NullActivity` object is as follows:

- No SBB entities are attached to the Activity Context of the `NullActivity` object, and
- No SLEE Facilities reference the Activity Context of the `NullActivity` object, and
- No events remain to be delivered on the Activity Context of the `NullActivity` object.

All methods that may change the attachments or references of an Activity Context are mandatory transactional. Hence, if an SBB method invocation changes an Activity Context's attachments and references, it must do so within a transaction. Therefore, when the transaction commits, the SLEE only has to examine the `NullActivity` objects whose Activity Contexts were enrolled in the transaction to determine which `NullActivity` objects should end.

For example, an SBB object creates a new `NullActivity` object within a transaction since the `createNullActivity` method of the `NullActivityFactory` interface is a mandatory transactional method (see Section 7.10.2). If the SBB object does not attach its assigned SBB entity (or another SBB entity available to it, e.g. a child SBB entity) to the Activity Context of the new `NullActivity` object or bind a name to the Activity Context of the new `NullActivity` object using the Activity Context Naming Facility, then when the transaction commits, the SLEE will end the new `NullActivity` object because the Activity Context of the `NullActivity` object will have no attachments and references. If the transaction rolls back, then the end result should be the same as if the `NullActivity` object was not created. *(Clarified in 1.1)*

A possible sequence of events and invocations for this example follows:

1. The SLEE starts a new transaction and invokes an event handler method of an SBB object with this transaction.
2. The event handler method creates a new `NullActivity` object by invoking a `NullActivityFactory` object's `createNullActivity` method.

   This enrolls the `NullActivity` object in the current transaction.

   The following sequence would also be the same if the event handler method detaches the only SBB entity attached to a `NullActivity` object's Activity Context and this Activity is not referenced by any SLEE Facilities.
3. The event handler method returns and the SLEE commits the current transaction successfully.
4. The SLEE determines that the `NullActivity` is not attached to any SBB entity, is not referenced by any SLEE Facility, and has no outstanding events remaining to be delivered on its Activity Context. It ends the `NullActivity` object.

   The `NullActivity` object may perform this determination. For example, the transaction coordinator may notify the `NullActivity` object that the transaction has committed by invoking a callback method on the `NullActivity` object. This callback method determines if the `NullActivity` object remains attached to at least one SBB entity or remains referenced by at least one SLEE Facility.
5. At this point, the Activity Context is in the Ending state and the SLEE does not allow additional events to be fired on the Activity Context.
6. As there are no outstanding events on the Activity Context to be delivered, the SLEE fires an Activity End Event on the Activity Context.
7. The SLEE performs the two following sequences concurrently or in an unspecified order:
   a. The SLEE delivers the Activity End Event to each SBB entity that is attached to the Activity Context and is interested in the Activity End Event (see Section 8.6.5). After the SLEE delivers the Activity End Event to the interested SBB entity, it detaches the interested SBB entity from the Activity Context.
   b. The SLEE detaches the SBB entities that are attached to the Activity Context and are not interested in the Activity End Event.

8. The SLEE transitions the Activity Context to the Invalid state and the SLEE can reclaim the Activity Context.

7.10.5.2 Explicitly ending a `NullActivity` object

An SBB object can also explicitly end a `NullActivity` object by invoking the `NullActivity` object's `endActivity` method. When the transaction in which the `endActivity` method was invoked commits, the `NullActivity` object ends.

A possible sequence of events and invocations for this example follows:

1. The SLEE starts a new transaction and invokes an event handler method of an SBB object with this transaction.
2. The event handler method ends a `NullActivity` object by invoking its `endActivity` method.
3. The `NullActivity` object notifies the SLEE that the `NullActivity` object has ended.
4. The SLEE marks the Activity Context of the `NullActivity` object as being in the Ending state.
5. The event handler method returns and the SLEE commits the current transaction successfully.

   If the transaction rolls back, then the `NullActivity` object will not end and the SLEE does not mark the Activity Context as being in the Ending state.
6. The SLEE continues to deliver any event outstanding on the Activity Context.
   At this point, the SLEE does not allow additional events to be fired on the Activity Context.
7. The last outstanding event on the Activity Context has been delivered. The SLEE fires an Activity End Event on the Activity Context.
8. The SLEE performs the two following sequences concurrently or in an unspecified order:
   a. The SLEE delivers the Activity End Event to each SBB entity that is attached to the Activity Context and is interested in the Activity End Event (see Section 8.6.5). After the SLEE delivers the Activity End Event to the interested SBB entity, it detaches the interested SBB entity from the Activity Context.
   b. The SLEE detaches the SBB entities that are attached to the Activity Context and are not interested in the Activity End Event.
9. The SLEE notifies SLEE Facilities that have references to the Activity Context that the Activity End Event has been delivered on the Activity Context. The Activity Context Naming Facility will remove the Activity Context's name bindings. The Timer Facility will remove any timers that depend on the Activity Context. Both of these Facilities will release their references to the Activity Context.
10. The SLEE transitions the Activity Context to the Invalid state and the SLEE can reclaim the Activity Context and the `NullActivity` object.

7.10.6 JNDI names of `NullActivity` related objects

The following table lists the names of the SLEE specification defined names for `NullActivity` related objects under `java:comp/env`.

Objects	***Names***
`NullActivityFactory` object	slee/nullactivity/factory
`NullActivityContextInterfaceFactory` object	slee/nullactivity/activitycontextinterfacefactory

7.11 Activity Context and Activity object persistence

All Activity Contexts are persistent. They have persistent state associated with them, e.g. Activity Context attributes. Their attachment (applies to SBB entities) and reference (applies to SLEE Facilities) relations are also persistent. However, the Activity object of the Activity Context may not be persistent. For example, a TCP connection will not survive a SLEE process failure. The SLEE together with the resource adaptor entity that owns the Activity object is responsible for ending the Activity object if the Activity object does not survive such a failure. *(Clarified in 1.1)*

All SLEE specification defined Activity objects are persistent. These Activity objects include `NullActivity`, `ProfileTableActivity` (see Section 10.9), and `ServiceActivity` (see Section 8.8.1) objects.

Chapter 8 Events

Events represent occurrences of significance or state transitions within the SLEE or outside the SLEE. They are used to convey information about the occurrence from one entity in the SLEE to another entity in the SLEE. These entities may be the SLEE, SBB entities, resource adaptor entities, and SLEE Facilities.

For example, the SLEE fires an Activity End Event on an Activity Context when the Activity Context's Activity object ends.

Events should be used for communication between SBB entities which are in different SBB entity trees. The Activity Context can be used to share common state. SBB Local interfaces are designed for communication between SBB entities within the same SBB entity tree.

8.1 Overview

Event producers fire events. Event consumers receive events. The SLEE event router delivers events fired by event producers to event consumers.

Each event fired by an event producer is represented within the SLEE by:

- An event object (see Section 8.1.5).
- An event type (see Section 8.1.7).

8.1.1 Event producers

Event producers include:

- Resource adaptor entities.
  Events fired by resource adaptor entities are also known as resource events.
- The SLEE.
  Events fired by the SLEE are known as SLEE events. The event types fired by the SLEE include the Activity End Event, the Profile Added Event, the Profile Removed Event and the Profile Updated Event, and Service Started Event.
- SLEE Facilities.
  Events fired by SLEE Facilities are known as SLEE Facility events. For example, the SLEE Timer Facility fires Timer Events.
- SBB entities.
  SBB objects in the Ready state, i.e. instances that have been assigned to SBB entities, may fire custom events types. Custom event types are event types defined by the SBB Developer. Typically, SBB entities use custom events to signal each other.

8.1.2 Event consumers

The only event consumers defined by the SLEE specification are SBB entities. The SLEE specification defines how events are delivered to SBB entities.

8.1.3 SBB as an event producer and fire event methods

Each SBB that fires events must declare the event types that it fires. For each event type that the SBB fires, the SBB Developer declares an abstract fire event method in the SBB abstract class. The abstract fire event method is implemented by the SLEE when the SBB is installed into the SLEE as part of a deployment unit.

An SBB object invokes a fire event method to pass an event to the SLEE event router. The event name binding for the event type that the fire event method fires determines the name of the fire event method (see Section 8.5.1).

8.1.4 SBB as an event consumer and event handler methods

Each SBB that receives events must declare the event types that it receives. For each event type that the SBB receives, the SBB defines and implements an event handler method in its SBB abstract class. This event handler method contains application logic to process events of the event type.

The SLEE event router invokes an event handler method of an SBB object to pass an event to the SBB entity for processing. The event name binding for the event type that the event handler method receives determines the name of the event handler method (see Section 8.5.2).

8.1.5 Event object

An event object is the Java object that represents the event and carries the information to be conveyed. For example, an `ActivityEndEvent` object represents an Activity End Event. An event object must be a `java.lang.Object`, i.e. it cannot be a primitive type.

8.1.6 Event class

The publicly declared Java class or interface of an event object is known as the event class. For example, the event class of a Timer Event is the `TimerEvent` interface, and not the concrete implementation class of this interface.

An event class must be defined in a named package, i.e. the class must have a `package` declaration. *(Added in 1.1)*

8.1.7 Event type

When an event producer fires an event, it must provide the event type of the event to the SLEE. The event type of the event completely determines how the SLEE will route the event, i.e. which SBB entities will receive the event, and which event handler method will be invoked to process the event.

The event type of an event is independent of its event class. The event producer defines the event types that it can fire in a deployment descriptor element. The event producer documentation, specification and deployment descriptor elements are expected to contain descriptive text that specifies how the event producer maps the events that the event producer fires to different event types.

For example, the Java Call Control specification defines the `JccCallEvent` event class and the `JccConnectionEvent` event class. Both `JccCallEvent` and `JccConnectionEvent` have an event identifier indicating the type of the event.

It is desirable to define a different event handler method for each event identifier code, rather than a different event handler for each event class. This allows the SBB Developer finer control over the event types that an SBB would like to process. For example, the SBB could choose to process Connection Alerting Events but not Connection Connected Events.

To allow this fine grain control over event routing, the Java Call Control resource adaptor type should provide a unique event type for each event identifier, i.e. all events with the same event identifier belong to the same event type. *(Clarified in 1.1)*

In the case of a User Location and Status resource adaptor type, providing an event type for each event class defined by the User Location and Status specification would be very appropriate. Each of the following event classes defined in a User Location and Status specification should have its own event type: `CamelReportResultEvent`, `CamelReportErrorEvent`, `EmergencyReportResultEvent`, `EmergencyReportErrorEvent`, and so on.

When an event producer fires an event, it provides the SLEE with the event type of the event either directly or indirectly (see Section 8.2.1). The SLEE does not inspect the event object or the Activity object on which the event was fired to determine the event type of the event.

The SLEE specification does not provide SBBs with any application code visible API for event types. SBB components cannot define methods with event type arguments and cannot invoke methods with event type

arguments. Instead SBBs are provided with a pattern where the event is referenced from the name of the method. Resource Adaptors identify event types as described in section 15.9.

An event type is identified by three strings - the event type's name, vendor and version. This representation of the event type is used in deployment descriptor elements. It is recommended that the SBB Developers should adopt the fully qualified Java package naming convention for event type names.

8.1.8 Custom event types

Custom event types are event types introduced by SBB Developers.

The primary difference between a custom event object and a non-custom event object is that custom event objects are created by SBB objects while non-custom event objects are created by other event producers.

All custom event objects must:

- Be serializable.
  The event class of a custom event type must specify either a Java class or interface that is serializable. This restriction allows a distributed SLEE implementation to distribute the custom event objects to multiple Java Virtual Machines.
- Implement `hashCode`.
  The actual implementation class of a custom event object must override the `Object.hashCode` method. The new `hashCode` method does not need to provide hash codes that are unique but does need to provide hash codes that are well distributed. A well-distributed hash code algorithm reduces but does not eliminate the number of times different custom event objects evaluate to the same hash code. When different custom event objects evaluate to the same hash code they are stored together and uniquely identified by their `equals` method. The custom event object overrides the `hashCode` method for the benefit of hash tables such as those provided by `java.util.Hashtable`. See `java.lang.Object` for more information.
- Implement `equals`.
  The actual implementation class of a custom event object must override the `Object.equals` method. The new `equals` method must ensure that the implementation of the method is reflexive, symmetric, transitive and for any non-null value `X`, `X.equals(null)` returns false. See `java.lang.Object` for more information.

8.1.9 Non-custom event types

Since the SLEE event objects and SLEE Facility event objects are created by the SLEE, these event objects already conform to the SLEE requirements for non-custom event objects. Event objects produced by resource adaptor entities must conform to the requirements described in Chapter 15.

8.1.10 Event name

An SBB refers to an event type by an event name. An event name is an SBB scoped local name that refers to an event type. This name must conform to the naming requirements of Java method names.

- The event name of an event type determines the name of the event handler method that receives events of the event type or the name of the fire event method used to fire events of the event type.
- An SBB uses the event name of an event type to identify the event types that should be masked (see Section 8.5.3).

8.2 Events and Activity Contexts

There are two important relationships between events and Activity Contexts. The two main relationships are:

- An event producer always *fires* an event on an Activity Context and an optional default address. The event producer adds an event to the event stream represented by the Activity Context. When

doing so, the event producer provides the event (consisting of the event object and event type), the Activity Context on which the event is fired on, and an optional default address.
Note: A resource adaptor entity uses the Activity Handle of an Activity Context to identify the Activity Context that an event is fired on (see Section 15.8).

- An SBB entity always *receives* an event on an Activity Context.
  The SBB entity receives an event from the event stream represented by the Activity Context. An SBB entity will only receive events on an Activity Context that it is attached to. The SLEE will not deliver an event fired on an Activity Context to an SBB entity that is not attached to the Activity Context. In the case of an initial event, the SLEE will attach the SBB entity to the Activity Context before delivering the initial event to the SBB entity (see Section 8.6).

8.2.1 Firing an event on an Activity Context and a default address

When an event producer fires an event, it must provide to the SLEE, either directly or indirectly, three pieces of information.

- The event.
  This includes the Java event object and the event type. A resource adaptor entity provides the event type directly when it passes the event to the SLEE (see Section 15.14.8), and an SBB object indirectly identifies the event type through the fire event method used to fire the event (see Section 8.5.1).
- The Activity Context that the event is fired on.
  The event producer identifies an Activity Context using one of the Activity Context's visible Java objects. An SBB object identifies an Activity Context by an Activity Context Interface object (see Section 7.4). A Resource Adaptor identifies an Activity Context by an Activity Handle object (see Section 15.8).
- An optional default address.
  The event producer may provide an optional default address. The default address is used to support certain standard initial event policies built into the SLEE (see Section 8.6). An event being fired by an event producer may have one or more addresses associated with it. These addresses may be obtained from the event object or from the Activity object. The event producer may provide one of these addresses as the default address.

  In general, the default address is obtained from an event class attribute or an Activity class attribute for all events of the same event class. A well-specified resource adaptor type should always identify the attribute that provides the default address for each event type that its resource adaptor entities fire.

  If the event type of the event is an initial event type of a Service and the initial event policy for the event type is a built-in policy that uses the default address, the SLEE event router uses the default address together with the Address Profile Table of the Service (see Section 10.24.1) to determine whether a new root SBB entity of the Service should be instantiated to process the event. If a new root SBB entity is instantiated, this event is the new SBB entity's initial event (see Section 8.6.1).

8.2.2 Receiving an event on an Activity Context

An SBB entity receives an event through one of its event handler methods (see Section 8.5.2). When the SLEE delivers an event to an SBB entity, it invokes the event handler method for the event's event type on an SBB object that is assigned to the SBB entity. The SLEE passes the following pieces of information to the event handler method:

- The event.
  The SLEE passes the event object (or a copy of the event object) handed to it by the event producer to the event handler method. The event type of the event determines which of the event han-

dler methods of the SBB abstract class is invoked by the SLEE to process the event. Hence, the event type can be inferred from the event handler method invoked.

- The Activity Context that the event is fired on.
  The SLEE passes to the event handler method an object implementing the SBB Activity Context Interface interface. The SBB entity uses this object to interact with the Activity Context on which the event was fired and the underlying Activity object of the Activity Context (see Section 7.1.1).

8.3 Events and Event Context

Added in 1.1: Event Context is new to 1.1.

Every event has an Event Context associated with it. The Event Context allows an SBB to suspend and resume delivery of the event associated with the Event Context. For more information on suspend and resume refer to 8.3.1.

An Event Context object is passed to an SBB object via the event handler method of the SBB (see Section 8.5.2). An SBB object can store an Event Context object in an SBB CMP field of type `javax.slee.EventContext`.

8.3.1 Event delivery suspend and resume

In SLEE 1.0 an event is considered to have been delivered to an SBB entity once the SBBs event handler method has returned. The implication is that the SBB has completed all of its work by the point it has returned from the event handler method. This requires the SBB to perform all event handling work in a synchronous manner, if other SBB entities are going to process the same event (according to event delivery priority, see section 8.6 particularly 8.6.5 through 8.6.8).

For efficiency reasons it is not always desirable to perform all work in a synchronous manner, particularly if it causes threads to be frequently blocked waiting for I/O. Therefore SLEE 1.1 introduces an API that allows an SBB entity to indicate that it has not finished its event handling work *even though it may have returned from its event handling method*.

This API allows an SBB to suspend event delivery, indicating that the SBB has not finished its event handling work. The SBB is then expected to use the API to resume event delivery at a later point, typically in another event handler method. Once delivery is resumed the SLEE continues to deliver the original event.

From the perspective of event delivery, if the delivery is suspended, the event is not delivered to a subsequent SBB entity, i.e. the event is considered to be "being delivered" to the SBB entity that suspended delivery. Once the event delivery is resumed it may be delivered to other SBB entities, i.e. once resumed the event is considered to have been processed by the SBB entity that suspended delivery.

Whilst the delivery of an event is suspended, other events fired on the Activity Context are not processed. This is to preserve the FIFO ordering of event processing, see 9.10.

In case of a programming error in the SBB, event delivery is only ever suspended for a time period. This time period can be provided by the SBB as an argument, or if not provided it is a platform specific default. The SLEE automatically resumes the delivery of the event after the period elapses.

Event delivery suspend/resume enables modeling of synchronous behavior in the SLEE asynchronous environment. For example an SBB can perform an asynchronous query of an external system, "waiting" for the response without blocking a thread. It achieves this by suspending event delivery before asynchrously querying an external system, then resuming event delivery in a subsequent transaction when handling the response from the external system.

8.3.2 `EventContext` interface

The `EventContext` interface defines the Java interface of an Event Context object and provides an SBB object with access to information and state about events and event delivery.

The public interface of the `EventContext` interface is as follows:

```
package javax.slee;
```

```
public interface EventContext {
        public Object getEvent();
        public ActivityContextInterface getActivityContextInterface();
        public Address getAddress();
        public ServiceID getService();
        public void suspendDelivery()
                throws  IllegalStateException,
                        TransactionRequiredLocalException,
                        SleeException;
        public void suspendDelivery(int timeout)
                throws  IllegalArgumentException,
                        IllegalStateException,
                        TransactionRequiredLocalException,
                        SleeException;
        public boolean isSuspended()
                throws  TransactionRequiredLocalException,
                        SleeException;
        public void resumeDelivery()
                throws  IllegalStateException,
                        TransactionRequiredLocalException,
                        SleeException;
}
```

8.3.2.1 `EventContext` interface `getEvent` method

This method returns the event object of the event that this Event Context is associated with.

8.3.2.2 `EventContext` interface `getActivityContextInterface` method

This method returns the Activity Context Interface for the Activity which the event was fired on. The returned Activity Context is the generic Activity Context Interface.

8.3.2.3 `EventContext` interface `getAddress` method

This method returns the address which was provided when the event was fired. This will be `null` if the address provided was `null`.

8.3.2.4 `EventContext` interface `getService` method

This method returns the component identifier of the Service provided when the event was fired. This will be `null` if no service component identifier was provided, or if the provided identifier was `null`.

8.3.2.5 `EventContext` interface `suspendDelivery` methods

These methods suspend the delivery of the event associated with this Event Context. Whilst event delivery is suspended and event is not delivered to other eligible SBB entities. For more information on the model refer to 8.3.1 An Event Context is only ever suspended for a specific period of time. The first variant of this method suspends the delivery of the event until some system specific default timeout is reached, while the second variant suspends delivery of the event for a specific timeout period in milliseconds. The timeout period is measured from the time the `suspendDelivery` method is invoked. Sometime after the timeout period expires[8], delivery of the event is automatically resumed. Event delivery can also be manually resumed by an SBB before the timeout period expires using the Event Context's `resumeDelivery` method.

[8] The actual suspended time will be at least as long as the timeout period, but may be longer for reasons such as scheduling restrictions, transaction dependencies, etc.

These methods are mandatory transactional methods. The event processing by the SLEE for this event is suspended if the enclosing transaction commits. If the transaction does not commit then the Event Context will not be "suspended". This method throws the following exceptions:

- The `java.lang.IllegalArgumentException` is thrown if the `timeout` argument is zero or a negative value.
- The `javax.slee.IllegalStateException` is thrown if the event delivery has already been suspended or `suspendDelivery` is invoked on an Event Context object that is not the Event Context object provided by the SLEE to the event handler method invoked by the SLEE. *Note: It is possible to get access to Event Context objects that are not provided by the SLEE as part of an event handler invocation, e.g. from a CMP field. These Event Context objects cannot be suspended by an application because they are not the Event Context of the "current" event, however they may be resumed.*
- The `javax.slee.TransactionRequiredLocalException` is thrown if this method is invoked without a valid transaction context.
- The `javax.slee.SleeException` is thrown if the event could not be suspended due to a system level failure.

8.3.2.6 `EventContext` interface `isSuspended` method

This method determines if the delivery of the event associated with this Event Context is suspended. This method returns `true` if the event delivery is suspended or `false` otherwise. This method is a mandatory transactional method.

This method throws the following exceptions:

- The `javax.slee.TransactionRequiredLocalException` is thrown if this method is invoked without a valid transaction context.
- The `javax.slee.SleeException` is thrown if the status of the event could not be determined due to a system level failure.

8.3.2.7 `EventContext` interface `resumeDelivery` method

This method resumes the delivery of the event associated with this Event Context. For more information on the model refer to section 8.3.1. This method is a mandatory transactional method. The event processing by the SLEE for this event is resumed if the enclosing transaction commits. If the transaction does not commit then the Event Context will not be "resumed" and therefore the event will not be delivered to any other eligible SBB entities[9].

- This method throws the following exceptions:
- The `javax.slee.IllegalStateException` is thrown if the Event Context is not in "suspended state", e.g. event delivery had not been suspended or event delivery has already been resumed.
- The `javax.slee.TransactionRequiredLocalException` is thrown if this method is invoked without a valid transaction context.
- The `javax.slee.SleeException` is thrown if the event could not be resumed due to a system level failure.

8.4 Event related deployment descriptor elements

The SLEE specification defines two sets of event-related deployment descriptor elements:

[9] Note that the SLEE may automatically resume suspended event contexts. For more information refer to 8.3.2.5.

- Deployment descriptor elements specified by event producers that declare the event types and event classes that the event producers fire.
- Deployment descriptor elements specified by SBB Developers that declare how SBBs use different event types.

8.4.1 Event producer deployment descriptor elements

Every event producer must identify the event types that it fires.

- An SBB declares each event type it fires in an `event` deployment descriptor element in its `sbb` deployment descriptor element.
- A resource adaptor type declares each event type its resource adaptor entities fire in an `event` deployment descriptor element in its `resource-adaptor-type` deployment descriptor element.
- A Resource Adaptor indirectly identifies the event types that it may fire through the `resource-adaptor-type-ref` deployment descriptor elements in its `resource-adaptor` deployment descriptor element. The SLEE assumes that for each resource adaptor type implemented by a resource adaptor, the Resource Adaptor may fire events of the event types referenced by that resource adaptor type.
- The event types of events fired by the SLEE and SLEE Facilities are defined by the SLEE specification. These event types must always be available to applications installed in the SLEE.

An `event` deployment descriptor element contains an `event-type-ref` element that references an `event-definition` element declared in another deployment descriptor. Each `event-definition` element defines an event type.

Each `event-definition` element contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- An `event-type-name` element, an `event-type-vendor` element and an `event-type-version` element.
  Together, these elements contain the human readable representation of the event type.
- An `event-class-name` element.
  This element provides the Java type for the event parameter of the event handler method that receives events of the specified event type or the fire event method that fires events of the specified event type. This means that all events of the same event type must extend a common base class or implement a common base interface. In the most generic case, the event class is `java.lang.Object`.

8.4.2 Event related SBB deployment descriptor elements

The SBB Developer must provide an `event` element for each event type that the SBB may receive or fire (see Section 8.1). Each `event` element contains the following attributes and sub-elements:

- An `event-direction` attribute.
  This attribute indicates if the SBB receives or fires the event type identified by the `event-type-ref` element of the `event` element. The value of this attribute can be “Receive”, “Fire”, or “FireAndReceive”.
- An `initial-event` attribute.
  This attribute indicates if the event type is an initial event (see Section 8.6.1). This attribute is meaningful only if the SBB is receiving the specified event type.
- A `mask-on-attach` attribute.
  This attribute indicates whether events of the event type identified by the `event-type-ref` element should be masked when an SBB entity attaches to an Activity Context. This attribute is

meaningful only if the event direction is “Receive” or “FireAndReceive”. By default, the event type is not masked.

- A `description` element.
  This is an optional informational element.
- An `event-type-ref` element.
  This element references an event type. It contains the following elements:
  - An `event-type-name` element, an `event-type-vendor` element, and an `event-type-version` element.
    These elements uniquely identify an event type declared in an `event-definition` element specified in another deployment descriptor. An `event-definition` element declares an event type (see Section 8.4.1)
- An `event-name` element.
  This element provides the SBB scoped name that the SBB uses to refer to the event type identified by the `event-type-ref` element (see Section 8.1.10). It is also used to compute the method names of the event handler method and the fire event method for the event type identified by the `event-type-ref` element.
- Zero or more `initial-event-select` elements.
  These elements are only meaningful if `initial-event` is true. They indicate which convergence name variables should be selected (see Section 8.6.3). It has the following attribute:
  - A `variable` attribute.
    The value of this attribute can be one of “ActivityContext”, “AddressProfile”, “Address”, “EventType”, or “Event”.
- An `initial-event-selector-method-name` element.
  This element is optional and is meaningful only if `initial-event` is true. It identifies an initial event selector method. The SLEE invokes this optional method to determine if an event of the specified event type is an initial event if the SBB is a root SBB of a Service (see Section 8.6.4).
- An `event-resource-option` element.
  This element is optional. It provides additional event handling options that the SLEE forwards to resource adaptor entities that emit events of the event type identified by the `event-type-ref` element. The SLEE does not interpret the content in this element. The event producer of the identified event type interprets this content.

  For example, the `event-resource-option` element destined for Java Call Control resource adaptor entities could specify whether “Notify” or “Block” behavior is desired on a `JccConnection` object (a `JccConnection` object is an Activity object) after a Java Call Control resource adaptor entity hands an event of the identified event type to the event router. If “Notify” is selected, then the `JccConnection` object can continue processing and additional events may be fired on the `JccConnection` object before this event has been completely processed by the SBB entities that can receive the event. If “Block” is selected, the `JccConnection` object cannot perform further processing and cannot generate additional events. The `JccConnection` object may continue processing after an SBB entity invokes the `continueProcessing` method on the `JccConnection` object.

  If different SBBs specify conflicting options, the event producer is responsible for resolving the conflict. In the above example, if an SBB specifies “Notify” and another specifies “Block”, the Java Call Control resource adaptor entity may resolve the conflict in favor of “Block”.

8.4.3 Feature interaction analysis

The SBB Developer may use the information present in the `event` elements to perform feature interaction analysis during service creation. Service creation tools may also aid feature interaction analysis by presenting this information in an SBB Developer friendly manner.

8.5 SBB abstract class and `SbbContext` interface event methods

The following methods of SBB abstract class of an SBB and the `SbbContext` interface interact with the SLEE event router.

- Fire event methods for each event type fired by the SBB.
- Event handler methods for each event type received by the SBB.
- The `SbbContext` interface `maskEvent` and `getEventMask` methods (see Section 8.5.3).

The following methods in the `ActivityContextInterface` interface, described earlier in Section 7.4, also interact with the SLEE event router.

- The `attach` method.
  This method attaches an SBB entity to an Activity Context. This allows the SBB entity to receive events fired on the attached Activity Context (see Section 8.6.5).
- The `detach` method.
  This method detaches an SBB entity from an Activity Context. This prevents the SBB entity from receiving events fired on the just detached Activity Context (see Section 8.6.5).

8.5.1 SBB abstract class fire event methods

Changed in 1.1: An additional fire event method has been added. This method allows the SBB to fire an event to a particular target Service on an Activity Context.

For each event type fired by an SBB, the SBB Developer must:

- Provide an `event` element in the SBB's `sbb` deployment descriptor element.
  The value of the `event-direction` attribute of the `event` element must be "Fire" or "FireAndReceive". The `event` element must also include an `event-name` element and an `event-type-ref` element. The `event-name` element provides the SBB scoped name used within the SBB abstract class to identify the event type and determines the name of the fire event method. The `event-type-ref` element references an `event-definition` element that provides the event type and the event class.
- Provide an `event-definition` element if the SBB introduces a new custom event type.
  If the custom event type that will be fired by the SBB does not already have an `event-definition` deployment descriptor element, the SBB Developer must provide an `event-definition` element for the newly introduced custom event type.
- Declare an abstract fire event method in the SBB abstract class.
  The SLEE implements this abstract method when the SBB is installed into the SLEE.

The name of the fire event method is derived from the event name of the event type that will be fired using the fire event method. The method name of the fire event method is derived by adding a "fire" prefix to the event name. The fire event method has one of the following method signatures:

```
public abstract void fire<event name>(<event class> event,
                                      ActivityContextInterface activity,
                                      Address address);
public abstract void fire<event name>(<event class> event,
                                      ActivityContextInterface activity,
                                      Address address,
                                      ServiceID service);
```

- The first method signature can be used if the SBB does not need to fire events to specific Services. It maintains backwards compatibility with the SLEE 1.0 specification.
- The second method signature allows an SBB to fire an event to a specific Service. If the `service` argument is not `null` then only SBB entities belonging to the specified Service are eligible to receive the event and only new root SBB entities belonging to the specified Service are eligible to be initiated by the event. If this argument is `null`, the behavior is the same as in the first method signature, i.e. SBB entities belonging to all Services are eligible to receive the event and new root SBB entities belonging to all Services are eligible to be initiated by the event. (See Section 8.6 for more details on event routing and other eligibility requirements.) *(Added in 1.1)*
- The fire event method must be declared as `public` and `abstract`.
- The fire event method does not have a `throws` clause.
- The `event` and `activity` arguments cannot be `null`. If either argument is `null`, the fire event method throws a `java.lang.NullPointerException`.
- The `address` argument provides the optional default address. If there is no default address, then it should be `null`. Refer to section 8.6 to learn how the default address influences event routing.
- As described earlier, an event producer fires an event on an Activity Context, an optional default address, and optionally to a specific Service. In the case of an SBB object as an event producer, the SBB object provides an Activity Context Interface object to identify the Activity Context, an `Address` object to provide the default address, and a `ServiceID` object to identify the Service that is the target for the event. The `address` argument must be `null` if there is no default address. See Section 8.6 to learn how the default address influences event routing. The `service` argument may be `null` if the event should be delivered to all interested SBBs in all active Services.
- The SBB object must be in the Ready state, i.e. it must have an assigned SBB entity, when it invokes this method. Otherwise, this method throws a `java.lang.IllegalStateException`.
- It is a mandatory transactional method (see Section 9.6.1). Hence, the SBB object must invoke this method within a transaction. It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

8.5.1.1 Restrictions

The SBB Developer must observe the following restrictions when developing an SBB that fires events.

- An SBB object cannot modify an event object after it passes the event object as an argument to a fire event method invocation.
- An SBB object cannot modify the event object even after the fire event method returns.
- An SBB object can only invoke its own fire event methods.
- An SBB object can only invoke fire event methods from within an execution thread used to deliver an event to an SBB object (either the same SBB object that is firing the event, or a different SBB object in the same SBB entity tree). *(Clarified in 1.1)*

The execution behavior of the SBB object and the SLEE is undefined if these restrictions are not followed.

8.5.1.2 Example

The following example illustrates an `event-definition` element, an `event` element, and a fire event method of a hypothetical Foo SBB that introduces and fires a new custom event type.

The example `event-definition` element is as follows:

```
        …
        <event-definition>
                <description>
                        An event of this type is fired when a FooActivity starts.
                        The event class of this event is com.foobar.FooEvent.
                        This event type identifies a com.foobar.FooEvent with
                        getCause() == START_EVENT.
                </description>
                <event-type-name> com.foobar.FooEvent.StartEvent </event-type-name>
                <event-type-vendor> com.foobar </event-type-vendor>
                <event-type-version> 1.3a </event-type-version>
                <event-class-name> com.foobar.FooEvent </event-class-name>
        </event-definition>
        …
```

The SBB deployment descriptor of the example Foo SBB is as follows:

```
<sbb>
        …
        <sbb-classes>
                <sbb-abstract-class >
                        …
                        <sbb-abstract-class-name>
                                com.foobar.FooSbb
                        </sbb-abstract-class-name>
                </sbb-abstract-class>
                …
        </sbb-classes>
        …
        <event event-direction=”Fire”>
                <description>
                        This SBB fires events of the com.foobar.FooEvent.StartEvent
                        event type and assigns StartEvent as the event name of
                        this event type.
                </description>
                <event-name> StartEvent </event-name>
                <event-type-ref>
                        <event-type-name> com.foobar.FooEvent.StartEvent </event-type-name>
                        <event-type-vendor> com.foobar </event-type-vendor>
                        <event-type-version> 1.3a </event-type-version>
                </event-type-ref>
        </event>
        …
</sbb>
```

The SBB abstract class of the example Foo SBB is as follows:

```
package com.foobar;
…
public abstract FooSbb implements javax.slee.Sbb … {
        …
        public abstract void fireStartEvent(com.foobar.FooEvent event,
                                            ActivityContextInterface activity,
                                            Address address);
        …
}
```

8.5.2 SBB abstract class event handler methods

For each event type received by an SBB, the SBB Developer must:

- Provide an `event` element in the SBB's `sbb` deployment descriptor element.
  The value of the `event-direction` attribute of the `event` element must be "Receive" or "FireAndReceive". It must also include an `event-name` element and an `event-type-ref` element. The `event-name` element provides the SBB scoped name used within the SBB abstract class to identify the event type and determines the name of the event handler method. The

`event-type-ref` element references an `event-definition` element that provides the event type and the event class. The `event-definition` element is provided and defined by the event producer of the event type. The `initial-event` attribute of the `event` element may optionally be set to "True". The `event` element may optionally include an `event-resource-option` element.

- Implement an event handler method in the SBB abstract class.
  This method contains the application logic that will be invoked to process events of this event type.

The name of the event handler method is derived from the event name of the event type that will be received by the event handler method. The method name of the event handler method is derived by adding an "on" prefix to the event name. The event handler method has one of the following method signatures:

```
public void on<event name>(<event class> event,
                           <SBB Activity Context Interface interface> activity);
public void on<event name>(<event class> event,
                           <SBB Activity Context Interface interface> activity,
                            EventContext eventContext);
```

- The first method signature without an Event Context argument is used if the SBB does not need access to Event Context. It maintains backwards compatibility with the SLEE 1.0 specification.
- The second method signature provides access to the Event Context associated with the event. *(Added in 1.1)*
- An SBB can only have one event handler method for each event type. *(Added in 1.1)*
- The event handler method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The event handler method is a mandatory transactional method (see Section 9.6.1). Hence, the SLEE always invoke this method within a transaction.
- As described earlier, an SBB object as an event consumer receives an event on an Activity Context. This Activity Context is the Activity Context on which the event was fired. In the case of an event handler method, an SBB Activity Context Interface object (if the SBB defines an SBB Activity Context Interface interface) or a generic Activity Context Interface object represents the Activity Context on which the event was fired.
- An Event Context is associated with the event that is fired if the SBB implements an event handler method that includes an Event Context argument. The Event Context can be used to suspend and resume further event processing of this event (refer Section 8.3.1). *(Added in 1.1)*
- The event handler method may return inadvertently by throwing a `RuntimeException`. See Section 6.9 and Section 6.10.1 on how the SLEE handles this situation.

8.5.2.1 Restrictions

Changed in 1.1: SBBs may modify event objects subject to restrictions, and Event Context objects may be used to store CMP references to event objects.

The SBB Developer must observe the following restrictions when developing an SBB that receives events.

- Event handler methods may modify event objects passed to it subject to the following restrictions:
  - Modifications are able to be marshaled (if the event is fired from a Resource Adaptor and the event was fired with the `SLEE_MAY_MARSHAL` flag set), or Java serialized if the event is a custom event.
  - Modifications may not change the event type of the event.
- After an event handler method invocation returns, the invoked SBB object cannot continue to hold a reference to the event object passed to the event handler method, either directly or indirectly through other objects. If an SBB entity needs access to an event object after an event handler

method for that event has returned, then the SBB can store the Event Context for that event in a CMP field and obtain the event object via this Event Context CMP field.

The execution behavior of the SBB object and the SLEE is undefined if these restrictions are not followed.

8.5.2.2 Example

The following example illustrates an `event-definition` element, an `event` element, and the event handler method of a hypothetical Foo SBB that receives an event type whose event type name is "com.some.event.types.PlayEvent".

The resource adaptor type's deployment descriptor contains the following element:

```
        …
        <event-definition>
                …
                <event-type-name> com.some.event.types.PlayEvent </event-type-name>
                <event-type-vendor> com.some </event-type-vendor>
                <event-type-version> 3.2.1a </event-type-version>
                <event-class-name> com.some.events.PlayEvent </event-class-name>
        </event-definition>
        …
```

The SBB's deployment descriptor is as follows:

```
<sbb>
        …
        <sbb-classes>
                <sbb-abstract-class>
                        …
                        <sbb-abstract-class-name>
                                com.foobar.FooSbb
                        </sbb-abstract-class-name>
                </sbb-abstract-class>
                …
                <sbb-activity-context-interface>
                        …
                        <sbb-activity-context-interface-interface-name>
                                com.foobar.FooSbbActivityContextInterface
                        </sbb-activity-context-interface-interface-name>
                </sbb-activity-context-interface>
                …
        </sbb-classes>
        …
        <event event-direction="Receive" initial-event="True">
                …
                <event-name> PlayEvent </event-name>
                <event-type-ref>
                        <event-type-name> com.some.event.types.PlayEvent </event-type-name>
                        <event-type-vendor> com.some </event-type-vendor>
                        <event-type-version> 3.2.1a </event-type-version>
                </event-type-ref>
                …
                <initial-event-select variable="AddressProfile"/>
                <initial-event-select variable="ActivityContext"/>
                …
        </event >
        …
</sbb>
```

In the FooSbb.java:

```
package com.foobar;
…
public abstract FooSbb implements javax.slee.Sbb {
        …
        private SbbContext sbbContext;
        …
        public void setSbbContext(SbbContext context) {
                …
```

```
                sbbContext = context;
                …
        }
        …
        public void onPlayEvent(com.some.events.PlayEvent event,
                                FooSbbActivityContextInterface ac) {
                …
                if (…) {
                        …
                        return;
                }
                …
                if (sbbContext.getRollbackOnly()) {
                        // do my own cleanup
                        …
                        return;
                }
        }
        …
}
```

8.5.3 `SbbContext` interface `maskEvent` and `getEventMask` methods

An SBB entity can manage the event types that it may receive for a particular Activity Context which it is attached to. This is known as *event masking*.

The `SbbContext` object of every SBB object has a `maskEvent` method and a `getEventMask` method.

8.5.3.1 `SbbContext` interface `maskEvent` method

The method signature for the `maskEvent` method is as follows:

```
        void maskEvent(String[] eventNames, ActivityContextInterface activity)
                        throws  NullPointerException,
                                TransactionRequiredLocalException,
                                IllegalStateException,
                                UnrecognizedEventException,
                                NotAttachedException,
                                SLEEException;
```

- The `eventNames` argument specifies the event names of the event types that should be masked. The set of event names specified must identify a subset of the event types that the SBB can receive, i.e. event names defined in `event` elements whose `event-direction` attribute is set to "Receive" or "FireAndReceive". A `null` or empty array argument unmasks all previously masked event types.
- The `activity` argument identifies the Activity Context whose events should be masked from the SBB entity assigned to the SBB object that invoked the `maskEvent` method, i.e. the specified event mask applies to a specific SBB entity and a specific Activity Context.
- The SBB object of the `SbbContext` object must be in the Ready state, i.e. it must have an assigned SBB entity, when it invokes this method. Otherwise, this method throws a `java.lang.IllegalStateException`.
- This method throws a `java.lang.NullPointerException` if the `activity` argument is `null`.
- This method throws a `javax.slee.UnrecognizedEventException` if one of the event names specified does not identify an event type that the SBB can receive.
- This method throws a `javax.slee.NotAttachedException` if the SBB entity is not attached to the specified Activity Context.

- The effects of the `maskEvent` method are not incremental or cumulative. When the `maskEvent` method is invoked twice with two different sets of event names, the second set overwrites the first set.
- When an SBB entity is initially attached or reattached to an Activity Context, the event types that are masked are the event types identified by `event` elements whose `mask-on-attach` attribute is set to "True" (see Section 8.4.2).
- An SBB entity can unmask or request the SLEE to resume the delivery of certain event types by invoking the same `maskEvent` method. It unmasks a set of event types by invoking the `maskEvent` method with a subset of the previously specified set of masked event names. It would exclude the event names of the event types that should be unmasked from the subset.
- The `maskEvent` method is a mandatory transactional method (see Section 9.6.1). Hence, the SBB object must invoke this method within a transaction. It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.
- The effects of an invocation of this method are not persistent and visible beyond the current transaction until after the transaction commits.

8.5.3.2 `SbbContext` interface `getEventMask` method

The method signature for the `getEventMask` method is as follows:

```
String[] getEventMask(ActivityContextInterface activity)
                throws  NullPointerException,
                        TransactionRequiredLocalException,
                        IllegalStateException,
                        NotAttachedException,
                        SLEEException;
```

- This method returns a set of event names. The SBB entity that is assigned to the SBB object of the `SbbContext` object uses this method to obtain the set of currently masked event types (referenced by their event names) for the Activity Context identified by the `activity` argument.
- The SBB object of the `SbbContext` object must be in the Ready state, i.e. it must have an assigned SBB entity, when it invokes this method. Otherwise, this method throws a `java.lang.IllegalStateException`.
- This method throws a `java.lang.NullPointerException` if the `activity` argument is `null`.
- This method throws a `javax.slee.NotAttachedException` if the SBB entity is not attached to the specified Activity Context.
- If the SBB entity has not invoked the `maskEvent` method to set an event mask after the SBB entity is attached or reattached to the Activity Context, then the set of event names returned by the `getEventMask` method is the set of event names of `event` elements whose `mask-on-attach` attribute is set to "True".
- Otherwise, the set of event names returned by the `getEventMask` method is the same as the set of event names specified by the most recent `maskEvent` method for the same SBB entity and same Activity Context.
- The `getEventMask` method is a mandatory transactional method (see Section 9.6.1). Hence, the SBB object must invoke this method within a transaction. It throws a `javax.slee.TransactionRequiredLocalException` if it is invoked without a valid transaction context.

- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

8.6 Event routing

The event routing subsystem of the SLEE performs the following functions:

- Instantiate a new root SBB entity of a Service to process events.
- Route an event to the SBB entities that have registered an interest in events of selected event types.
- Deliver events according to the child relation event priority order.

8.6.1 Initial events

An event that causes the SLEE to instantiate a new root SBB entity of a Service to receive the event is known as an initial event. The initial event types of the root SBB of a Service identify the event types that may cause the SLEE to instantiate a new root SBB entity of the Service. They are also known as the initial event types of the Service.

The set of initial event types of an SBB is a subset of the set of event types that can be received by the SBB. This is because the SBB must have an event handler method to receive the events of an initial event type. More precisely, the initial event types of an SBB are the event types referenced by `event` elements within the SBB's `sbb` element that have their `event-direction` attribute set to "Receive" or "FireAndReceive", and their `initial-event` attribute set to "True".

A potential root SBB must have at least one `event` attribute with the `initial-event` attribute set to "True". However, not all potential root SBBs become a root SBB of an active Service. Since the SLEE only instantiates root SBB entities for an active Service, the `initial-event` attribute is only meaningful when the SBB is playing the root SBB role of an active Service.

When an event producer fires an event on an Activity Context, the following algorithm is used to determine whether the event is an initial event for each active Service.

- Determine the set of active Services whose set of initial event types includes the event type of the event.
- For each Service in this set,
    - Compute a convergence name. See Section 8.6.2 below for how this name is computed. A convergence name is a name used to uniquely identify a root SBB entity belonging to a Service.
        - If an initial event selector method is provided and it indicates that this event is not an initial event for this Service (see Section 8.6.4), then the next step is to start at the beginning of the Service loop and perform initial event processing for the next Service in the set.
    - Look for a root SBB entity of the Service with the convergence name.
    - If no such root SBB entity is found, then the event is an initial event for a new root SBB entity of the Service.
        - Create a new root SBB entity of the Service.
          If the creation of the root SBB entity fails, then the following two steps are skipped. The creation of the root SBB entity may fail because the `sbbCreate` method or the `sbbPostCreate` method returned by throwing an exception (e.g. a `RuntimeException` or a `CreateException`), or because of SLEE-level or system-level problems. The SLEE should log this occurrence.
        - Assign the convergence name to the new root SBB entity of the Service.

        - Attach the new root SBB entity to the Activity Context on which the event was fired.
    - Otherwise, a root SBB entity is found.
        - If the root SBB entity found is not already attached to the Activity Context on which the event was fired, then attach the root SBB entity found to the Activity Context.

When the SBB entity is being attached to the Activity Context, the SLEE event router sets the event mask of the Activity Context with respect to the SBB entity according to the `mask-on-attach` attributes of the `event` deployment descriptor elements of the SBB.

Once the new or existing SBB entity is attached to the Activity Context, the root SBB entity will receive the event when the SLEE event router delivers the event to the SBB entities attached to the Activity Context (see Section 8.6.8 below).

8.6.2 Computing the convergence name

Logically, the convergence name is a canonical name constructed by concatenating the following variables. A variable is either selected or not selected (see Section 8.6.3). If a variable is not selected, its value is `null`. Otherwise, its value is as specified below.

- Activity Context identifier.
  This value of this variable (if selected) is the identifier that uniquely identifies an Activity Context within the SLEE. This value is SLEE implementation specific and is known only to the SLEE. There are no SBB visible APIs to obtain or interact with Activity Context identifiers.
- Address Profile identifier. *(Clarified in 1.1)*
  The value of this variable (if selected) is the unique name of an Address Profile in the Address Profile Table of the Service. The `addresses` attribute of this Address Profile is matched against a "selected" address. The selected address is either the address provided when the event was fired (if no initial event selector method was invoked), or the address provided by the initial event selector method. The initial event selector method may leave the default address unchanged, or may specify an alternative address.

  If the selected address is `null`, then the value of this variable is `null`. If the selected address is not `null` and an Address Profile is found for this address, then the value of this variable is the name of the Address Profile. If the selected address is not `null` and an Address Profile cannot be found for the address, then the SLEE does not construct a convergence name at all for the event and the event is not an initial event for the Service.

  A default address, or the address returned by an initial event selector method, can only be present in at most one Address Profile in an Address Profile Table. Hence, a single address maps to at most one Address Profile identifier and the SLEE constructs at most one convergence name for the at most one Address Profile identifier that the address is present in.

  Future releases of the SLEE specification are likely to allow a single address to map to multiple Address Profiles (and Profile identifiers). In this case, the SLEE will construct a convergence name for each Address Profile identifier that the address maps to.

  The default address and the selection of the Address Profile identifier in the convergence name play an important role in the creation of a convergence name. Examples where a matching Address Profile might not be found are if the default address is not present in the `addresses` attribute of any of the profiles in the Address Profile Table, or if the Address Profile Table for the Service does not exist in the SLEE.

- Address.
  The value of this variable (if selected) is the default address passed to the event router when the event was fired by the event producer or returned by the initial event selector method. If the address is `null`, then the value of this variable is `null`.
- Event type.
  The value of this variable (if selected) is the event type of the event that was fired.
- Event.
  The value of this variable (if selected) is unique for each event fired, e.g. each invocation of an SBB fire event method or each firing of an event by a resource adaptor (using SLEE vendor specific fire event methods). It is unique regardless of whether the same pair of event object and Activity Context object (in the case of SBB fired event, or Activity object in case of resource adaptor fired event) are passed to the fire event method. There are two unique events fired in the following scenarios:
  - An SBB invoking the same fire event method twice. From the SLEE's perspective, the two fire method invocations fire two unique events even if the Activity Context object and event object passed to the fire event method are the same.
  - An SBB firing an event in its event handler method. The event fired through the fire event method is a different event even if the same Activity Context object and event object passed to the event handler method is passed to the fire event method.
  - A resource adaptor entity invoking one or more SLEE provided methods for firing events multiple times. From the SLEE's perspective, these invocations fire multiple unique events even if the Activity object and event object passed are the same.
- Custom name.
  If the root SBB of the Service identifies an initial event selector method for the event type, then the value of this variable is provided by this method. Otherwise, this variable is not selected.

Two convergence names are equal if the values of each of these variables in the two convergence names are equal.

8.6.3 Selecting convergence name variables

The SLEE provides two ways to select convergence name variables for each initial event type. An SBB may choose to combine the two methods as desired or required. *(Clarified in 1.1)*

- Variable selection by deployment descriptor elements.
  The variables are selected by the `initial-event-select` deployment descriptor elements in the `event` deployment descriptor element of each initial event type.
  - Each `initial-event-select` element has a single mandatory `variable` attribute. The value of this attribute can be one of the following:
    - "ActivityContext" to select the Activity Context identifier variable.
    - "AddressProfile" to select the Address Profile identifier variable.
    - "Address" to select the address variable.
    - "EventType" to select the event type variable.
    - "Event" to select the event variable.
  - The custom name variable is never selected. Only the initial event selector method can provide a custom name.
- Variable selection by initial event selector method.
  The variables are selected at runtime by the initial event selector method.
  - The `event` element must contain an `initial-event-selector-method-name` element. The `initial-event-selector-method-name` element identifies the

initial event selector method for this event type in the SBB abstract class. Multiple initial event types may share the same initial event selector method.
It is expected that an initial event selector method will be defined for the event types emitted by each kind of Activity.

- The `event` element may contain `initial-event-select` elements. These elements provide selection hints to the initial event selector method. Ultimately, the initial event selector method determines which convergence name variables are selected.
- The custom name variable is selected if the custom name value returned by the initial event selector method is not `null`.

8.6.4 Initial event selector method

Clarified in 1.1: The 1.0 specification was inconsistent with the 1.0 Java interface.

The method signature of the initial event selector method is as follows:

```
public InitialEventSelector
                <initial event selector method name>(InitialEventSelector ies);
```

- The initial event selector method must be declared as `public` and cannot be `static` , `abstract`, or `final`.
- The method name must not begin with “sbb” or “ejb”, and must be a valid Java identifier.
- This method is a non-transactional method.
- It is only invoked on SBB objects in the Pooled state.

The public interface of the `InitialEventSelector` interface is as follows:

```
package javax.slee;

public interface InitialEventSelector {
       /* attributes for selecting convergence name variables */
       public boolean isActivityContextSelected();
       public void setActivityContextSelected(boolean select);
       public boolean isAddressProfileSelected();
       public void setAddressProfileSelected(boolean select);
       public boolean isAddressSelected();
       public void setAddressSelected(boolean select);
       public boolean isEventTypeSelected();
       public void setEventTypeSelected(boolean select);
       public boolean isEventSelected();
       public void setEventSelected(boolean select);

       /* the event type, event name, event object, Activity object, default address, and
          custom name */
       public String getEventName();
       public Object getEvent();
       public Object getActivity();
       public Address getAddress();
       public void setAddress(Address address);
       public String getCustomName();
       public void setCustomName(String customName);

       /* control attribute */
       public boolean isInitialEvent();
       public void setInitialEvent(boolean initial);
}
```

Before the SLEE invokes the initial event selector method, it initialises the `InitialEventSelector` object argument to the initial event selector method as follows:

- The `<variable>Selected` attributes are initialized to true if the `event` element contains an `initial-event-select` element that selects the convergence variable.

- The `eventName` and `event` attributes are initialized to according to the event fired by the event producer. The `eventName` attribute is initialized with the SBB defined event name corresponding to the event type of the event. The `event` attribute is initialized with the event object provided by the event producer or an event object logically equivalent to the event object provided by the event producer.
  The `InitialEventSelector` object may be instantiated in a different JVM than the event producer. The event object may be a remote copy of the original event object provided by the event producer.
- The `activity` attribute is initialized to the underlying Activity object of the Activity Context.
- The `address` attribute is initialized to the default address provided by the event producer. If the event producer did not provide a default address, this attribute is set to `null`.
- The `customName` attribute is set to `null`.
- The `initialEvent` attribute is set to `true`.

The implementation of the initial event selector method must observe the following rules:

- The method may use the set accessor methods of the `InitialEventSelector` object to modify the `InitialEventSelector` object.
- It must not modify the event object or the Activity object returned by the `getEvent` method and `getActivity` method. If it modifies these objects, then the execution behavior is unspecified.
- It must return the `InitialEventSelector` object passed in as its argument.
- The method name must not begin with "sbb" or "ejb", and must be a valid Java identifier.

When the initial event selector method returns, the SLEE computes the convergence name as described in Section 8.6.2 according to the information in the `InitialEventSelector` object.

- If the `initialEvent` attribute is set to `false`, then the event is not an initial event for this Service and no further convergence name and initial event processing is performed for this Service, i.e. the SLEE will not create any new root SBB entity for this Service to process this event. If this attribute is set to `true`, then this event is an initial event if the computed convergence name is unique to the Service.
- The selected convergence name variables are determined by the <*`variable`*>`Selected` attributes of the `InitialEventSelector` object.
- The `address` attribute of the `InitialEventSelector` object provides the default address. The value of this attribute may be `null` if there is no default address. The value of this attribute determines the value of the address variable in the convergence name if the address variable is selected and is also used to look up Address Profiles in the Address Profile Table of the Service if the Address Profile variable is selected.
  - If the `address` attribute is `null` when the initial event selector method returns, then the address convergence name variable is not selected, i.e. same as setting the `addressSelected` attribute to `false`.
  - If the `addressProfileSelected` variable is set to `true` and the address is not `null` but does not locate any Address Profile in the Address Profile Table of the Service, then no convergence name is created, i.e. same as setting the `initialEvent` attribute to `false`.
- If the value of `customName` attribute of the `InitialEventSelector` object is `null`, then the custom name variable is not selected.

8.6.5 SBB entities

After the SLEE event router performs the above procedure to instantiate new root SBB entities and to attach root SBB entities to the Activity Context, the SLEE event router begins to deliver the event to the SBB entities that may receive the event.

An SBB entity may receive the event if:

- The SBB entity is attached to the Activity Context on which the event was fired, and
- The SBB entity has an event handler method for the event type of the event, and
- The SBB entity has not masked the event type of the event on the Activity Context.

8.6.6 Concurrency control

The SLEE event router provides the following concurrency control semantics:

- The SLEE event router guarantees that there will be no concurrent execution within a single SBB object. The SLEE will invoke all the methods of an SBB object serially. These methods include the event handler methods, the local interface methods, and the life cycle callback methods
- The SLEE event router guarantees that two or more SBB entities will not concurrently receive events on any single Activity Context. This allows the SBB Developer to develop event handler methods without having to consider the implications of multiple execution threads that may interact with the same Activity Context and the same underlying Activity object.
- Since an event is always fired on an Activity Context, the SLEE event router will always deliver the event serially to SBB entities that may receive the event. The order in which these SBB entities receive the event depends on their event delivery priorities (see Section 8.6.7).

8.6.7 Event delivery priority

The event delivery priority of an SBB entity is determined in the following way:

- The `default-priority` element of the `service` deployment descriptor element of a Service specifies the initial event delivery priority of the root SBB entities instantiated by the SLEE for the Service. An SBB can modify the event delivery priority of a root SBB entity at runtime by invoking the `setSbbPriority` method on an SBB local object that represents the root SBB entity. *(Clarified in 1.1)*
- The `default-priority` element in the `get-child-relation-method` element of the `sbb` deployment descriptor element of a parent SBB specifies the initial event delivery priority of the child SBB entities created on this child relation. A parent SBB entity can modify the event delivery priority of a child SBB entity at runtime by invoking the `setSbbPriority` method on an SBB local object that represents the child SBB entity.

The event delivery priority of an SBB entity is an integer in the range of -128 through 127, inclusive. -128 is the lowest priority and 127 is the highest priority. The following guidelines for assigning event delivery priorities should be followed by the SBB Developer and/or Service Deployer to convey an SBB's sensitivity to execution order and priority relative to its siblings.

- 100 to 127 for highest event delivery priority.
  Child SBB entities of the same parent SBB entity with the highest event delivery priority should receive the event first.
- 31 to 99 for high event delivery priority.
  Child SBB entities of the same parent SBB entity with the high event delivery priority should receive events earlier than child SBB entities of the same parent SBB entity with standard event delivery priority.
- -30 to 30 for standard event delivery priority.
  Child SBB entities of the same parent SBB entity with the standard event delivery priority should

receive events earlier than child SBB entities of the same parent SBB entity with low event delivery priority.

- -99 to –31 for low event delivery priority.
  Child SBB entities of the same parent SBB entity with the low event delivery priority should receive the event later than child SBB entities of the same parent SBB entity with standard event delivery priority.
- -128 to -100 for lowest priority.
  Child SBB entities of the same parent SBB entity with the lowest event delivery priority should receive the event last.

8.6.8 Event delivery semantics

The SLEE delivers an event fired on Activity Context in the following way:

- The SLEE delivers the event to a parent SBB entity before it delivers the event to the child SBB entities of the parent SBB entity.
- Among sibling SBB entities with the same parent, the SLEE delivers the event to a sibling SBB entity with a higher priority before it delivers the event to a sibling SBB entity with a lower priority.
- If the sibling SBB entities have child SBB entities, the SLEE delivers the event to the child SBB entities of the higher priority sibling SBB entity after it delivers the event to the higher priority sibling SBB entity and before it delivers the event to the lower priority sibling SBB entity. Hence, the event delivery order is higher priority sibling, children of higher priority sibling, lower priority sibling, then children of lower priority sibling.
- The order in which an event is delivered to sibling SBB entities with the same event delivery priority is not specified, i.e. the SLEE implementation determines the order.
- When the SLEE delivers an event to an SBB entity, it checks the event mask associated with the SBB entity and the Activity Context. If the event is masked, the SLEE does not invoke the event handler method for the event type of the event. Even though the SBB entity did not receive the event, the SLEE considers the event delivered to the SBB entity.
- After the SLEE delivers the event to an SBB entity, it determines if one or more SBB entities that were not attached to the Activity Context prior to delivering the event to the SBB entity have become attached to the Activity Context. If so, the SLEE must deliver the same event to these newly attached SBB entities.
- The SLEE delivers the event to an SBB entity that stays attached once. The SLEE may deliver the event to the same SBB entity more than once if it has been detached and then re-attached. For example:
  - An SBB entity that detaches before it receives the event will not receive the event.
  - An SBB entity that received the event in the past but became detached and then re-attached will receive the event again if it remains attached and the event is not masked.

8.7 Automatic event subscription

The SLEE must implement automatic event subscription. Automatic event subscription performs the following functions:

- Initial event subscription on behalf of active Services (see Section 8.7.1).
- Received event subscription on behalf of SBB entities (see Section 8.7.2).

The SLEE collaborates with resource adaptor entities to automatically install event triggers and filters into the underlying resources to subscribe and unsubscribe events. These event filters allow the resource adaptor entities and underlying resources to install the appropriate event triggers and filters close to the event source and thus avoid transporting a large number of unwanted events to the SLEE.

Refer to Section 15.12.4 for a description of the callback methods the SLEE invokes on resource adaptor entities in order to support automatic event subscription. *(Added in 1.1)*

8.7.1 Initial event subscription

The SLEE provides a standard mechanism to help a Service set event filters to subscribe for the initial events of the potential root SBB entities of the Service, (i.e.also known as the initial events of the Service).

The root SBB of a Service specifies the initial event types of the Service. The Address Profile Table of the Service (if specified) specifies the subscribed address set of the Service. The Service's subscribed address set is the set of addresses in the `addresses` attribute of all Address Profiles in the Address Profile Table of the Service. The initial event set of a Service is the cross product of the initial event types of the Service and the Service's subscribed address set. The SLEE's initial event set is simply the union of the initial event sets of all active Services, i.e. Services in the Active state, running in the SLEE.

The initial event set of the SLEE may change as a result of administrative actions. These administrative actions include: *(Clarified in 1.1)*

- Starting and stopping a Service.
  The SLEE will only instantiate root SBB entities for active Services. When a Service becomes active, the SLEE may have to expand the SLEE's initial event set to include the initial event set of the newly active Service. When a Service is deactivated, the SLEE has to contract the SLEE's initial event set to exclude the initial event set of the deactivated Service.
- Adding an Address Profile to the Address Profile Table of an active Service, removing an Address Profile from the Address Profile Table of an active Service, and modifying the addresses in an Address Profile in the Address Profile Table of an active Service.
  These actions indirectly change the SLEE's initial event set by changing the subscribed address set of an active Service.

8.7.2 Received event subscription

In addition to the SLEE's initial event set, the SLEE must also ensure that SBB entities running in the SLEE receive events of their received event types (which also includes the initial event types). The received event set of the SLEE is the set of events that all SBB entities running in the SLEE should receive.

The received event set of the SLEE changes when an operation that modifies the set of events that an SBB entity can receive is invoked. These operations include:

- Attaching an SBB entity to an Activity Context.
  This occurs implicitly when the SLEE instantiates a new SBB entity to handle an initial event of the SBB (see Section 8.6) or explicitly when the `attach` method is invoked (see Section 7.4.1). When this occurs, the SLEE may have to expand the SLEE's received event set to include events of the received event types of the SBB entity. An optimized SLEE implementation may expand the SLEE's event set to include only the events of the received event types of the SBB entity that could be fired by the underlying Activity object of the Activity Context.
- Detaching an SBB entity from an Activity Context.
  This occurs implicitly after the SLEE delivers an Activity End Event to an SBB entity (see Section 7.3.3), when the SBB entity is removed (see Section 2.2.6), or explicitly when the `detach` method is invoked (see Section 7.4.3). When this occurs, the SLEE may have to contract the SLEE's received event set to exclude events of the received event types of the SBB entity.
- Modifying an SBB entity's event mask on an Activity Context.
  This occurs implicitly when an SBB entity invokes the `maskEvent` method with a different set of masked event types. When this occurs, the SLEE may have to contract the SLEE's received event set if more events types are masked, or the SLEE may have to expand the SLEE's received event set if fewer event types are masked.

8.8 Service Started Event and `ServiceActivity` objects

Changed in 1.1: A new Service Started Event type is defined. This event is fired only to the starting service. The 1.0 Service Started Event type is deprecated.

The Administrator uses the SLEE's management interface to activate and deactivate Services. Each Service has an associated Activity of type `javax.slee.serviceactivity.ServiceActivity`.

The SLEE fires a Service Started Events on a Service's `ServiceActivity` after the Service is started. A Service is started after one of the following occurs:

- The SLEE is in the Running state and a `ServiceManagementMBean` object is invoked to activate the Service, or
- The SLEE becomes active when the SLEE transitions from the Starting to the Running state and the SLEE starts the Service because the persistent state of the Service says it should be active.

SBBs register interest in Service Started Events like any other events using an `event` element in their deployment descriptors and by attaching to the appropriate Activity Contexts. An SBB can only receive a Service Started Event fired by the SLEE as an initial event since this event is fired only once on a `ServiceActivity` object, at the time the corresponding Service Activity starts.

The Service Started Events enable an SBB to be notified when a Service becomes active. For example, the root SBB of a Service may listen to Service Started Events by specifying the event type of the Service Started Event as an initial event type of the SBB. The root SBB of a Wake-up Service may re-instate timers from persistent data stored in Profile Tables. Daemon-like Service entities may also use the Service Started Events and Service Activities to keep them alive instead of creating and attaching to `NullActivity` objects.

In the SLEE 1.0 specification, a Service Started Event could be delivered to the root SBB of any Service, not just of the Service that is starting. For example, consider the case where the root SBBs of both Service A and Service B listen to the Service Started Event as an initial event, and both of these Services are initially Inactive. When Service A is activated, a Service Started Event is fired on the Service Activity for Service A and a root SBB entity of Service A is created to receive the event. If Service B is then activated, a root SBB entity of Service B is created to receive the event, but the existing root SBB entity (or a new SBB entity, depending on the convergence name) of Service A will receive the same event also, as the Service Started Event is an initial event for both Services.

In the SLEE 1.1 specification, a new version of the Service Started Event is defined. When fired by the SLEE, this version of the event is delivered to the root SBB of the activating Service only (if the root SBB has an event handler method for the event type). The event is not delivered to SBBs in any other Service that may be listening for the event type. *(Added in 1.1)*

A 1.1-compliant SLEE implementation may need to fire both the 1.0 and 1.1 versions of the Service Started Event when a Service is activated, depending on the SBBs that have been installed in the SLEE and the event types that they have event handler methods for.

- In order to maintain backwards compatibility, the 1.0 version of the Service Started Event still needs to be delivered to all interested SBBs in all Services.
- The 1.1 version of the Service Started Event is only delivered to interested SBBs in the activating Service, regardless of whether SBBs in other Services have registered interest in the same event type.

8.8.1 `ServiceActivity` objects

The SLEE fires a Service Started Event on a `ServiceActivity` when the corresponding Service starts.

- The SLEE is the creator and owner of all `ServiceActivity` objects. A `ServiceActivity` object is created for a Service prior to firing a Service Started Event on the Activity.
- At any point in time, a Service has at most one `ServiceActivity` object.

- The SLEE makes a `ServiceActivityFactory` object accessible via the JNDI component environment of SBBs. The `ServiceActivityFactory` object provides the `ServiceActivity` object of the Service of the invoking SBB object. The SLEE may have creates this object if does not already exist.

The `ServiceActivity` interface is as follows:

```
package javax.slee.serviceactivity;

import javax.slee.ServiceID;

public interface ServiceActivity {
	public ServiceID getService();
}
```

- The `getService` method returns the Service identifier of the `ServiceActivity` object's Service. *(Added in 1.1)*

8.8.2 Ending a `ServiceActivity` object

A `ServiceActivity` object ends only when the corresponding Service is being stopped by one of the following occurances:

- A `ServiceManagementMBean` object is invoked to deactivate the Service.
- The SLEE transitions from the Running state to the Stopping state (implicitly stopping all Services).

8.8.3 Service Started Events

Changed in 1.1: A new version of the Service Started Event type has been added. This event is fired only to the starting Service.

The `event-definition` element for Service Started Events is as follows:

```
<event-definition>
	…
	<event-type-name>
		javax.slee.serviceactivity.ServiceStartedEvent
	</event-type-name>
	<event-type-vendor> javax.slee </event-type-vendor>
	<event-type-version> 1.0 </event-type-version>
	<event-class-name>
		javax.slee.serviceactivity.ServiceStartedEvent
	</event-class-name>
	<event-type-name>
		javax.slee.serviceactivity.ServiceStartedEvent
	</event-type-name>
	<event-type-vendor> javax.slee </event-type-vendor>
	<event-type-version> 1.1 </event-type-version>
	<event-class-name>
		javax.slee.serviceactivity.ServiceStartedEvent
	</event-class-name>
</event-definition>
```

The `ServiceStartedEvent` interface is as follows:

```
package javax.slee.serviceactivity;

import javax.slee.ServiceID;

public interface ServiceStartedEvent {
	public ServiceID getService();
}
```

- The `getService` method returns the Service identifier of the Service that has started. *(Added in 1.1)*

8.8.4 `ServiceActivityFactory` interface

Changed in 1.1: JNDI location constant added.

The SLEE specification defines the `ServiceActivityFactory` as a Java interface that contains a single `getActivity` method. The `ServiceActivityFactory` object is bound to a SLEE specification defined location in the component environment of every SBB component (see Section 7.10.6).

```
package javax.slee.serviceactivity;

import javax.slee.FactoryException;
import javax.slee.TransactionRequiredLocalException;

public interface ServiceActivityFactory {
        // JNDI location constant
        public static final String JNDI_NAME =
                "java:comp/env/slee/serviceactivity/factory";

        // methods
        public ServiceActivity getActivity()
                throws TransactionRequiredLocalException, FactoryException;
}
```

- The `JNDI_NAME` constant. *(Added in 1.1)*
  This constant specifies the JNDI location where a `ServiceActivityFactory` object may be located by an SBB component in its component environment.
- The `getActivity` method takes no arguments. It is the responsibility of the SLEE to provide an appropriate implementation for the SBBs in each Service. The implementation must return a `ServiceActivity` object for the Service the invoking SBB entity exists in.
- This method is a mandatory transactional method (see Section 9.6.1). It throws a `javax.slee.TransactionRequiredException` if it is invoked without a valid transaction context.
- This method throws a `javax.slee.FactoryException` if the method cannot obtain a `ServiceActivity` object for the Service because of system-level or SLEE-level problems. This exception is derived from `javax.slee.SLEEException`.

8.8.5 `ServiceActivityContextInterfaceFactory` interface

Changed in 1.1: JNDI location constant added.

An SBB can obtain a generic Activity Context Interface object of a Service Activity using a `ServiceActivityContextInterfaceFactory` object. The `ServiceActivityContextInterfaceFactory` object is bound to a SLEE specification defined location in the component environment of every SBB.

```
package javax.slee.serviceactivity;

import javax.slee.ActivityContextInterface;
import javax.slee.FactoryException;
import javax.slee.UnrecognizedActivityException;
import javax.slee.TransactionRequiredLocalException;

public interface ServiceActivityContextInterfaceFactory {
        // JNDI location constant
        public static final String JNDI_NAME =
                "java:comp/env/slee/serviceactivity/activitycontextinterfacefactory";

        // methods
        public ActivityContextInterface
                getActivityContextInterface(ServiceActivity serviceActivity)
                        throws  NullPointerException,
                                TransactionRequiredLocalException,
                                UnrecognizedActivityException,
                                FactoryException;
```

```
}
```

- The `JNDI_NAME` constant. *(Added in 1.1)*
  This constant specifies the JNDI location where a `ServiceActivityContextInterface-Factory` object may be located by an SBB component in its component environement.
- See Section 7.6.1 for a general description on Activity Context Interface Factories, the transactional semantics, and the common exceptions thrown by a `getActivityContext-Interface` method.

8.8.6 JNDI names of `ServiceActivity` related objects

The following table lists the names of the SLEE specification defined names for `ServiceActivity` related objects under `java:comp/env`.

Objects	***Names***
`ServiceActivityFactory` object	slee/serviceactivity/factory
`ServiceActivityContextInterface-Factory` object	slee/serviceactivity/activitycontextinterfacefactory

Chapter 9 Transactions

The previous chapters described the desired behavior of the SLEE and the desired behavior of the components hosted by the SLEE during normal failure-free execution. However, a SLEE running in a real world environment will encounter failures, such as Java Virtual Machine failures, process failures, host failures, and resource failures. This chapter defines how the SLEE applies well-established transaction concepts to address unexpected failures during execution.

All SLEEs must implement the transactional semantics defined in this specification.

Added in 1.1: There are two modifications to the transaction model which are introduced by this version of the specification. These are additional flexibility in the local transaction model and the definition of a transaction interface which resource adaptor objects may use.

9.1 Benefits

Transactions are a proven technique for simplifying application development. The SLEE's transactional semantics free the application developer from dealing with many of the issues of failure-recovery and concurrent execution. It allows the developer to design an application as if it were run in an environment that executes units of work serially with a defined failure model.

9.2 Atomicity

When a transaction executes, it may read and modify execution state directly and may invoke methods that directly or indirectly read and modify the execution state of the SLEE. The execution state is further partitioned into *transactional execution state* and *non-transactional execution state*. Transactional execution state is state that is associated with a particular transaction and is part of any commit or rollback of the transaction. Non-transactional execution state is state that is not directly associated with a particular transaction. An example of transactional execution state is SBB CMP attributes. An example of non-transactional execution state is a variable on a thread's stack.

Atomicity guarantees that when a transaction completes, all changes to the transactional execution state are applied (committed) or the transactional execution state is restored to its previous value (the transaction is rolled back). This is often referred to as "all or none" semantics.

The mechanism by which a compliant SLEE implementation provides atomicity guarantees is not defined by this specification.

9.3 Isolation

All transactions are isolated from each other at the serializable isolation level. This allows multiple units of work to be executed in parallel provided the end state is guaranteed to be reachable from at least one serial execution order. This also means changes to the transactional execution state of one transaction are not visible to other concurrently executing transactions.

There are cases where it may be impossible to maintain an isolation guarantee. These cases are covered by Section 9.11.

The mechanism by which a compliant SLEE implementation provides the isolation guarantee is not defined by this specification. Examples include the use of an underlying relational database that provides support for the required isolation, or a local lock manager to serialize access to transactional execution state. These examples are illustrative rather than prescriptive.

9.4 No demarcation API for application components

The SLEE specification does not define an API that an SBB or Profile can use to explicitly define or demarcate transactions. SBB objects can request that transactions be rolled back through the use of the `setRollbackOnly` method. Resource Adaptor objects have access to a transaction API, this API is discussed in Section 9.14.

9.5 Transaction state machine

Changed in 1.1: Updated to include how resource adaptor objects interact with transactions.

The SLEE starts a transaction prior to a SLEE originated invocation sequence (see Section 9.8 for a more complete description of a SLEE originated invocation sequence). On successful completion of the SLEE originated invocation sequence the SLEE attempts to commit the transaction.

A resource adaptor object may also start and commit or rollback transactions for its own use *(Added in 1.1).*

Each transaction in the SLEE has an implicit state machine. This state machine determines how each transaction should behave upon failures, successful or unsuccessful completion of a SLEE originated invocation sequence or resource adaptor object sequence.

A transaction may be in one of the following states:

- Active state
- Committing state
- Committed state
- Marked for Rollback state
- Rolling Back state
- Rolled Back state

The SLEE handles all transitions between the various states of the transaction finite state machine. An SBB object may directly cause a transition between the Active and Marked for Rollback states by invoking the `setRollbackOnly` method on its `SbbContext` object.

A resource adaptor object may directly cause a transition between the Active and Marked for Rollback states by invoking the `setRollbackOnly` method on the `SleeTransaction` object for the transaction *(Added in 1.1)*.

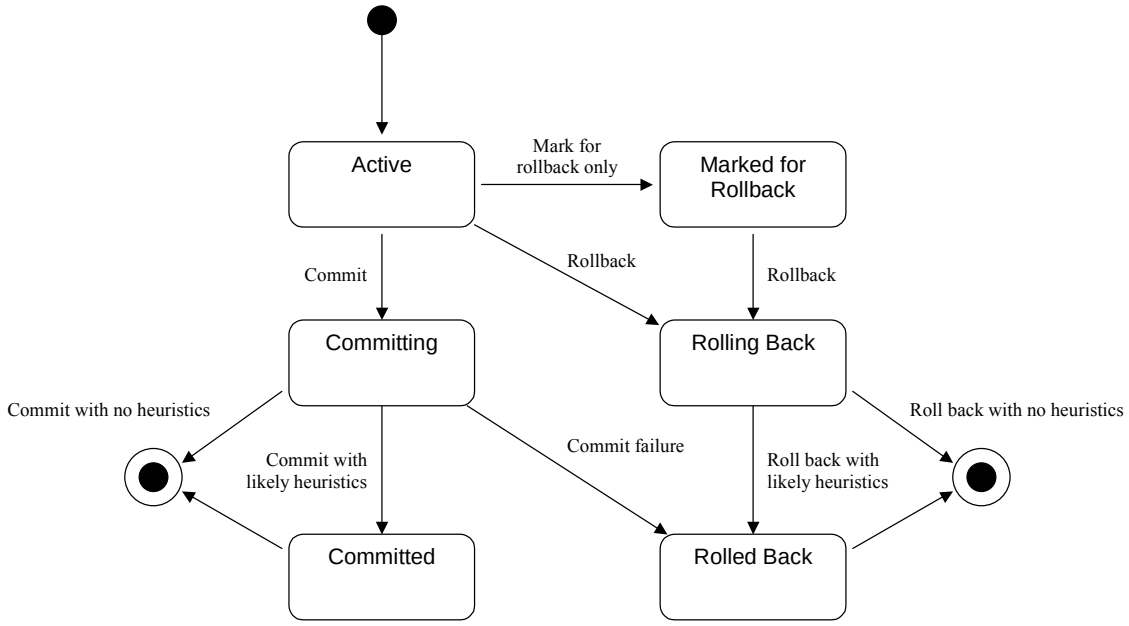

Figure 10 Transaction state transition diagram

9.5.1 Active state

A transaction enters this state on initialization. A transaction is typically created and initialized prior to the SLEE originated invocation sequence. A transaction may also be created by a resource adaptor object when

it needs to access transactional state, e.g. Profiles, when an existing transaction does not exist *(Added in 1.1)*. When in the Active state, the transaction context is propagated to any invoked transactional methods (see Section 9.6 for a list of transactional methods).

9.5.2 Committing state

On successful completion of a SLEE originated invocation sequence the transaction transitions from the Active state to the Committing state. For a transaction created by a resource adaptor object, this transition occurs when the resource adaptor object attempts to commit the transaction using either the `SleeTransaction` or `SleeTransactionManager` interface (see Section 9.14). In the Committing state, the transaction is in the process of being committed. If the commit succeeds the transaction either transitions to the end state or, if heuristics are present, transitions to the Committed state. Otherwise, it enters the Rolled Back state.

9.5.3 Committed state

A transaction enters this state from the Committing state when a heuristic has likely been applied as part of the commit. If a heuristic has not been applied the transaction would cease to exist. The SLEE specification does not define the semantics of transaction heuristics in the Committed state. For more information on possible transaction heuristics refer to JTA.

9.5.4 Marked for Rollback state

The Marked for Rollback state is an extension of the transaction life cycle provided by the SLEE. When a transaction is in this state the SLEE guarantees that any changes made to the transactional execution state will be rolled back. A transaction may enter this state from the following state:

- From the Active state:

  A transaction enters this state from the Active state when

  - The SBB invocation returns abnormally, i.e. it throws a `RuntimeException`. In this case, the SLEE marks the transaction for rollback. For further discussion see Section 9.12.
  - The SBB invocation directly or indirectly invokes the `setRollbackOnly` method.
  - A transactional method of a SLEE Facility, resource, or EJB marks the transaction for rollback.
  - The `setRollbackOnly` method is invoked by a resource adaptor object on a `SleeTransaction` object. *(Added in 1.1)*

9.5.5 Rolling Back state

A transaction in this state is in the process of rolling back. It may enter this state from the following states:

- From the Marked for Rollback state.
- From the Active state. The SLEE may rollback a transaction due to internal failures. In these cases, the SLEE originated invocation sequence or resource adaptor object sequence may have successfully completed execution, e.g. the methods within the SLEE originated invocation sequence returned without throwing a `RuntimeException`.

9.5.6 Rolled Back state

The transaction has been aborted, i.e. the transaction execution state has been restored to its original value. It may enter this state from the following states:

- From Rolling Back state.
  The transaction enters this state from the Rolling Back state when the transaction has rolled back but the transaction processing indicates that a heuristic has likely been applied as part of the roll-

back. If a heuristic has not been applied the transaction would cease to exist. The SLEE specification does not define the semantics of transaction heuristics in the Rolled Back state. For more information on possible transaction heuristics refer to JTA.

- From the Committing state.
  The transaction enters this state from the Committing state when the SLEE is unable to successfully commit the transaction. Any updates to the transactional execution state are rolled back on the transition between the Committing and Rolled Back states. SBB handling for this case is described in Section 6.10. The transaction may fail to commit for various reasons for example underlying resource failures, concurrency control failures, locking or deadlock failures.

9.6 Transactional methods

A transactional method is a method that may directly or indirectly read or modify transactional execution state. Transactional methods defined by the SLEE specification are further divided into *mandatory transactional methods* and *required transactional methods*.

- Mandatory transactional methods are methods that must be invoked from within a transaction. If they are not invoked from within a transaction the SLEE must throw a `javax.slee.TransactionRequiredLocalException`.
- Required transactional methods are methods that may or may not be invoked from within a transaction. If they are not invoked from within a transaction, a new transaction is created for the duration of the method invocation.

Both mandatory and required transactional methods execute in the same transaction context as the calling method.

When a method of a transactional resource or EJB is invoked, this method may or may not be invoked with a transactional context corresponding to the transaction of the SLEE originated invocation sequence. Whether this context is propagated depends on the SLEE implementation and the EJB container implementation. For further discussion, see Section 9.9.

9.6.1 Mandatory transactional methods

Changed in 1.1: Methods introduced in 1.1 which are mandatory-transactional have been added to this section.

These methods may only be invoked from within a transaction in the Active or Marked for Rollback state otherwise the SLEE must throw a `javax.slee.TransactionRequiredLocalException`.

- SBB abstract class `sbbCreate`, `sbbPostCreate`, `sbbRemove`, `sbbLoad`, and `sbbStore` life cycle methods.
  These methods are also known as the *mandatory transactional life cycle methods*.
- SBB abstract class event handler and `sbbRolledBack` methods.
  The SLEE cannot invoke any other event handler, or `sbbRolledBack` method with the same transaction.
- SBB abstract class CMP field get and set accessor methods.
  SBB CMP fields are part of the transactional execution state.
- SBB abstract class fire event methods.
  The fire event method of an SBB may be invoked to fire an event that will be processed asynchronously. When an SBB invokes a fire event method the SLEE stores the event internally as part of its internal transaction processing. The event will be delivered to other SBB entities at some later time if the transaction successfully commits. If the transaction does not commit successfully the SLEE discards the event. Each event handler method invoked to process an event fired asynchronously executes within a separate transaction. As other SBB entities will receive the fired event after the originating transaction has committed, transactional state changes made by the originating

transaction are visible to subsequent event handler method invocations, e.g. state held within an Activity Context.

- SBB local interface methods.
  The methods of an SBB local interface execute in the calling methods transaction context. For example if an SBB entity invokes a method on an SBB local interface and the called SBB entity invokes its `setRollbackOnly` method, then after the invocation returns the caller's `getRollbackOnly` returns `true`.
- Logical cascading removal methods implemented by the SLEE for each SBB (see Section 9.8.1 for more details).
- `ChildRelation` methods.
  The methods of a `ChildRelation` object execute in the caller's transaction context. Any transactional state changes caused by child SBB entity creation and removal are viewed as having taken effect within the calling transaction and only take effect if the calling transaction commits.
- `SbbContext` interface `setRollbackOnly` and `getRollbackOnly` methods.
- `ActivityContextInterface` interface `attach` and `detach` methods.
  The effects of these methods are considered transactional, and the state changes caused by these methods are part of the transaction they are invoked within.
- `SbbContext` interface `maskEvent` and `getEventMask` methods.
  The effects of these methods are considered transactional, and take effect as part of the successful commit of the transaction they are invoked within. Regardless of whether the underlying resource adaptor entity that fires the specified event types is transactional, the SLEE and the resource adaptor entity must collaborate to maintain the atomicity guarantee of the transactional programming model.
- `SbbContext` interface `getActivities` and `getSbbLocalObject` methods.
- Generic `ActivityContextInterface` interface methods.
- SBB Activity Context Interface interface attribute get and set accessor methods.
  Activity Context attributes are part of the transactional execution state.
- `NullActivity` interface methods.
  When a `NullActivity` object's `endActivity` method is invoked to end the `NullActivity` object, this method invocation executes within the current transaction. It only takes effect when the transaction that the `endActivity` method is invoked from commits successfully. See Section 7.10 for details on ending `NullActivity` objects.
- `NullActivityFactory` interface methods.
- `ServiceActivityFactory` interface methods.
- All Activity Context Interface Factory interface methods.
- `ActivityContextNamingFacility` interface methods.
  Bindings/unbindings made via the `ActivityContextNamingFacility` are transactional, and take effect as part of the successful commit of the transaction they execute within.
- `ProfileTable` interface methods. *(Added in 1.1)*
- Profile Local interface methods. *(Added in 1.1)*
  The methods of a Profile local interface execute in the calling methods transaction context.
- `SleeEndpoint` interface `startActivityTransacted`, `enrollInTransaction`, `fireEventTransacted`, and `endActivityTransacted` methods. *(Added in 1.1)*
- The `getActivityContextInterface` method on the `TimerFacility` interface. *(Added in 1.1)*

- Transactional methods of transactional resources (see Section 9.9).

9.6.2 Required transactional methods

A required transaction method is a transactional method that may or may not be invoked from within a transaction in the Active or Marked for Rollback states.

If a required transactional method is invoked from within a transaction, the calling method's transaction context is propagated to the called method.

If a required transactional method is invoked from a method that is executing outside of a transaction context, a new transaction is started and associated with the method during its invocation. The SLEE attempts to commit this transaction when the method returns. Once the transaction has committed or rolled back the calling method resumes execution.

Required transactional methods include:

- All `TimerFacility` interface methods, except the `getResolution` method, the `getDefaultTimeout` method and the `getActivityContextInterface` method. *(Changed in 1.1)*
  The setting and cancelling of timers is transactional, the state changes associated with these operations are part of the transaction the methods execute within.
- All `ProfileFacility` interface methods.

If a required transactional method is called from a method that does not have an associated transaction and the newly created transaction fails to commit, the calling method receives a `javax.slee.TransactionRolledbackLocalException`.

9.7 Non-transactional methods

Changed in 1.1: Methods introduced in 1.1 that are non-transactional have been added to this section.

A non-transactional method is a method that must not directly or indirectly read or modify transactional execution state. Non-transactional methods include:

- SBB abstract class `setSbbContext`, `unsetSbbContext`, `sbbActivate` and `sbbPassivate` life cycle methods.
- SBB abstract class initial event select method.
- `SbbContext` interface `getService`, `getSbb`, and `getTracer` methods.
- All get SBB Usage Parameter methods (see Section 11.4.1).
- All methods of non-transactional Facilities.
- All methods on the `ResourceAdaptor` and `ResourceAdaptorContext` interfaces.
- `SleeEndpoint` interface `startActivity`, `fireEvent`, and `endActivity` methods.
- All methods of non-transactional resources (see Section 9.11).
- Non-transactional methods of transactional resources.

A non-transactional method is always invoked in an unspecified transaction context.

9.7.1 Unspecified transaction context

The term "an unspecified transaction context" is used in the SLEE specification to refer to the cases in which the SLEE specification does not fully define the transaction semantics of a method invocation. The SLEE specification does not prescribe how a SLEE implementation should manage the execution of a method with an unspecified transaction context, i.e. the transactional semantics are left to the SLEE implementation.

When the transactional semantics of an unspecified transaction context is unknown, a method that runs with an unspecified transaction context must be written conservatively and not rely on any particular transaction semantics.

A failure that occurs during the execution of a method that runs with an unspecified transaction context may leave any resource, transactional and non-transactional state accessed from the method in an unpredictable state. The SLEE specification does not define how the application should recover state after such a failure.

9.8 SLEE and non-SLEE originated method invocations

The SLEE may directly or indirectly invoke transactional methods:

- Direct method invocation, also known as *SLEE originated method invocation*:
  The SLEE directly invokes a transactional method if there is no other SBB Developer implemented method invocation, such as an SBB abstract class method, and no other logical cascading removal method invocation (see below) in the call-chain leading to the current method invocation.
- Indirect method invocation, also known as *non-SLEE originated method invocation*.
  The SLEE indirectly invokes a transaction method if there is at least one other SBB Developer implemented method, such as an SBB abstract class method, or at least one other logical cascading removal method invocation in the call-chain leading to the current method invocation.

9.8.1 Special considerations for cascading removal operations

The SLEE implements a logical cascading removal method for each SBB. The SLEE invokes the logical cascading removal method of an SBB to perform a cascading removal operation on an SBB entity tree or SBB entity sub-tree whose root SBB entity is an instance of the SBB.

- Logical cascading removal methods are mandatory transactional methods.
- When a logical cascading removal method is invoked, the method invocation may cause additional transactional methods to be invoked. These additional method invocations are always considered to be non-SLEE originated method invocations. The transactional methods that may be invoked include the life cycle methods of the SBB entities in the SBB entity tree or the SBB entity sub-tree being removed.

A logical cascading removal method invocation may be a SLEE originated method invocation or a non-SLEE originated method invocation.

- A SLEE originated logical cascading removal method invocation is initiated by the SLEE to reclaim an SBB entity tree after the attachment count of the root SBB entity of the SBB entity tree decreases to zero. When the method invocation completes, an entire SBB entity tree should be removed.
  The SLEE specification does not specify whether or not the SLEE invokes the logical cascading removal method in the same transaction that decreases the attachment count of the root SBB to zero. Thus, a SLEE may implement eager and/or lazy reclamation of SBB entity trees. If the cascading removal operation does not occur in the same transaction (i.e. reclamation is lazy), the SLEE must behave as if the SBB entity tree has become non-existent. It must remove any convergence name associated with the root SBB entity of the SBB entity tree, i.e. the same convergence name must cause the SLEE to instantiate a new root SBB entity. It must also not deliver events to SBB entities in the "non-existent" SBB entity tree. *(Clarified in 1.1)*
- A non-SLEE originated logical cascading removal method invocation is initiated by an SBB to remove an SBB entity tree by invoking the `remove` method on an SBB local object, or by invoking one of the remove methods on a `ChildRelation` object, or by invoking the `remove` method on an `Iterator` object obtained from a `ChildRelation` object. *(Clarified in 1.1)*

In summary, the only SLEE originated method invocation in a cascading removal operation is the SLEE invoking a logical cascading removal method to remove an SBB entity tree after the attachment count of the root SBB entity of the SBB entity decreases to zero. Non-SLEE originated logical cascading removal

method invocations and all additional transactional methods invoked within a SLEE or a non-SLEE originated cascading removal method invocation are considered to be non-SLEE originated method invocations.

9.8.2 SLEE originated invocation

A *SLEE originated invocation* consists of all the transactional method invocations invoked directly by the SLEE within the same transaction on either a single SBB object or a single SBB entity. Each SLEE originated invocation must include at least one SLEE originated mandatory transactional method invocation.

The possible types of SLEE originated invocations are as follow:

- Life Cycle Only.
  - The possible SLEE originated method invocations within a SLEE originated invocation of this type are as follows:
    - at least one mandatory transactional life cycle method invocations, followed by
    - at most one `sbbExceptionThrown` method invocation.
  - Within a SLEE originated invocation, these methods must be invoked on the same SBB object, also known as the target SBB object of the SLEE originated invocation.
  - If the target SBB object enters the Ready state, then the target SBB entity of the SLEE originated invocation is the SBB entity represented by the SBB object.
- Remove Only.
  - The possible SLEE originated method invocations within a SLEE originated invocation of this type are as follows:
    - a SLEE originated logical cascading removal method invocation.
  - The target SBB entity of a Remove Only SLEE originated invocation is the root SBB entity of the SBB entity tree being removed. There is no target SBB object in a Remove Only SLEE originated invocation.
- Op Only.
  - The possible SLEE originated method invocations within a SLEE originated invocation of this type are as follows:
    - zero or more mandatory transactional life cycle method invocations, followed by
    - an invocation of either an event handler method or an `sbbRolledBack` method, followed by
    - zero or more mandatory transactional life cycle method invocations, followed by
    - at most one `sbbExceptionThrown` method invocation.
  - Within a SLEE originated invocation, these methods must be invoked on the same SBB object, also known as the target SBB object of the SLEE originated invocation.
  - If the SBB object enters the Ready state, then the target SBB entity of the SLEE originated invocation is the SBB entity represented by the SBB object.
  - The SBB object must enter the Ready state for the SLEE to invoke its event handler method or its `sbbRolledBack` method.
- Op And Remove.
  - The possible SLEE originated method invocations within a SLEE originated invocation of this type are as follows:
    - zero or more mandatory transactional life cycle method invocations, followed by
    - an invocation of either an event handler method or an `sbbRolledBack` method, followed by
    - zero or more mandatory transactional life cycle method invocations, followed by

        - at most one `sbbExceptionThrown` method invocation, followed by
        - a logical cascading removal method invocation.
    - Within a SLEE originated invocation, these methods, except the logical cascading removal method invocation, must be invoked on the same SBB object, also known as the target SBB object of the SLEE originated invocation. The SBB object must be in the Ready state when its event handler method or its `sbbRolledBack` method is invoked.
    - The target SBB entity of a SLEE originated invocation of this type is the SBB entity represented by the target SBB object.
      The target SBB entity is always known since the target SBB object must be in the Ready state.
    - The SBB entity tree or SBB entity sub-tree to be removed by the logical cascading removal method invocation within a SLEE originated invocation of this type must be the SBB entity tree or SBB entity sub-tree rooted by the target SBB entity.

If any of the methods (except the `sbbExceptionThrown` method) invoked on the target SBB object of a SLEE originated invocation returns by throwing a `RuntimeException`, then the SLEE invokes the `sbbExceptionThrown` method of the target SBB object of the transaction within the same transaction.

There is at most one `sbbExceptionThrown` method invocation in each of the above SLEE originated invocation types since the SLEE will move the target SBB object of the SLEE originated invocation to the Does Not Exist state after invoking the `sbbExceptionThrown` method once on the SBB object. The SLEE will not invoke any other methods on the SBB object once it enters the Does Not Exist state.

Each of the SLEE originated mandatory transactional method invocations may in turn invoke other methods.

- If an invoked method is a transactional method (mandatory or required), then the invoked method runs in the same transaction context as the SLEE originated mandatory transactional method invocation.
- If an invoked method is a non-transactional method, then the invoked method runs with an unspecified transaction context (see Section 9.7.1). *(Clarified in 1.1)*

9.8.3 SLEE originated invocations and transactions

Changed in 1.1: The 1.1 specification permits multiple SLEE originated invocation sequences within a single transaction. SLEE implementations need not implement multiple SLEE originated invocation sequences within a single transaction. The 1.1 semantics as defined below only applies to 1.1 Services and SBBs. A 1.1 compliant JAIN SLEE must implement and maintain the 1.0 contract for 1.0 Services and SBBs.

Each transaction initiated by the SLEE must be described by zero or more valid SLEE originated invocation sequences. A SLEE originated invocation sequence is an ordered sequence of one or more SLEE originated invocations. The SLEE completes a SLEE originated invocation in the ordered sequence before beginning the next SLEE originated invocation in the ordered sequence.

The following rules apply to valid sequences.

- A valid sequence contains at most one Op Only or Op And Remove SLEE originated invocation.
- A valid sequence contains at most one Remove Only or Op And Remove SLEE originated invocation.
    - If the sequence contains a Remove Only or an Op And Remove SLEE originated invocation, then the Remove Only or an Op And Remove SLEE originated invocation must be the last SLEE originated invocation in the sequence.
    - If the sequence contains one Op Only and one Remove Only SLEE originated invocations, then the target SBB object invoked by the Op Only SLEE originated invocation must enter the Ready state or be in the Ready state, and the target SBB entity of the Re-

move Only SLEE originated invocation must be the root SBB entity of the target SBB entity invoked by the Op Only SLEE originated invocation.

- A valid sequence does not contain SLEE originated method invocations on more than one SBB object that enters the Ready state from the Pooled state or is already in the Ready state.

  A SLEE originated logical cascading removal method invocation does not directly invoke any methods on an SBB object. The method directly invoked is the logical cascading removal method. The logical cascading removal method is logically a SLEE implemented method for each SBB. The SLEE must invoke this method on an SBB entity to initiate a cascading removal of the SBB entity tree rooted by the SBB entity. Logically, this method implements the cascading removal operation and is only invoked by the SLEE on the root of the SBB entity tree (i.e. not invoked on the descendent SBB entities in the SBB entity tree). An invocation of this method on the root of the SBB entity tree is considered a SLEE originated invocation. All `sbbRemove` *callbacks, and any local interface method calls invoked by an* `sbbRemove` *callback, are not considered to be directly invoked by the SLEE and are not SLEE originated invocations.*

  When a SLEE originated `sbbPostCreate` *method invocation returns without throwing a* `RuntimeException`*, the sequence is considered to have invoked an SBB object that enters the Ready state from the Pooled state*

9.8.4 Invocation of multiple SBB entity event handler methods in a single SLEE initiated transaction

Added in 1.1

By default the SLEE will only invoke one event handler method per SLEE initiated transaction. This is to ensure that lower priority SBB entities do not cause higher priority SBB entities transacted state changes to be rolled back. In some cases it is desirable to deliver an event to several SBB entities within a single transaction. A SLEE implementation may choose to invoke multiple event handler methods within a single transaction. The `last-in-transaction` attribute (see section 3.1.8) is used to influence transaction boundaries used by the SLEE when delivering an event to multiple SBB entities.

If an SBB event handler method has the `last-in-transaction` attribute set to "True"[10], then the SLEE must complete the current transaction before invoking any other event handler method. Once the transaction has been completed the SLEE may continue delivery of the event.

If an SBB event handler has the `last-in-transaction` attribute set to "False", then a SLEE implementation may, at its discretion, invoke the next lower priority SBB entity event handler method in the same transaction. The SLEE may continue delivering the event to subsequent lower priority SBB entities in the same transaction until either an event handler method specifies `last-in-transaction` as "True", or there are no more SBB entities to deliver to; in either case the SLEE will complete the transaction.

9.8.5 Examples of valid SLEE originated invocation sequences

The following list describes several example valid SLEE originated invocation sequences:

- This example sequence does not invoke any SBB entities.

  The SLEE directly invokes a single SBB objects in the Pooled state in this sequence. There is a Life Cycle Only SLEE originated invocation in the sequence and the target SBB object invoked remains in the Pooled state.

- This example sequence invokes a single SBB entity.

  The SLEE directly invokes a single SBB object that represents the SBB entity. There is a single SLEE originated invocation in the sequence. The types of the SLEE originated invocation in the sequence can be:

[10] The default value of this optional attribute is "True".

  - Life Cycle Only if the SBB object enters the Ready state, or
  - Op Only.

- This example sequence invokes a root SBB entity and a logical cascading removal method to remove the root SBB entity.

  The SLEE directly invokes a single SBB object that represents the root SBB entity. One of these invocations causes the attachment count of the root SBB entity to decrease to zero. The SLEE directly invokes the appropriate logical cascading removal method to remove the SBB entity tree. There is a one SLEE originated invocation in the sequence and the type of the only SLEE originated invocation in the sequence is Op and Remove.

- This example sequence invokes a single non-root SBB entity and a logical cascading removal method to remove the root SBB entity of the non-root SBB entity.

  The SLEE directly invokes a single non-root SBB object that represents a non-root SBB entity. One of these invocations causes the attachment count of the root SBB entity of the same SBB entity tree to decrease to zero. The SLEE directly invokes the appropriate logical cascading removal method to remove the SBB entity tree. There are two SLEE originated invocations in the sequence. The types of the first SLEE originated invocation in the sequence can be:

  - Life Cycle Only if the SBB object enters the Ready state, or
  - Op Only.

  The type of the second SLEE originated invocation in the sequence is Remove-Only.

9.8.6 An illustrated example of a transaction

Figure 11 illustrates an example transaction which contains one SLEE originated invocation sequence. The SLEE originated invocation sequence consists of a single Op Only SLEE originated invocation. In this example, the event is an initial event for the SBB. When this occurs, the SLEE creates an SBB entity and assigns it to an SBB object. The SLEE invokes the `sbbCreate` and `sbbPostCreate` life cycle methods, the event handler method for the event, and the `sbbStore` life cycle method with the same transaction. In this example the event processing completes successfully, i.e. without throwing a `RuntimeException`.

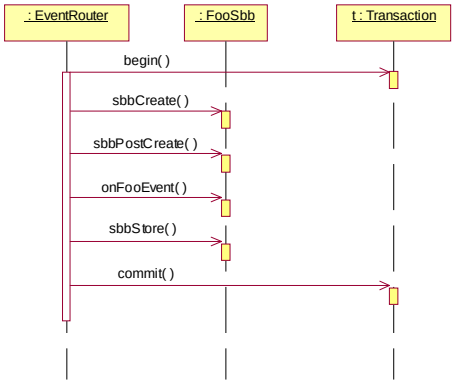

Figure 11 Transactional behavior of an SBB invocation that returns normally

9.9 Transactional resources and EJB invocations

There are two types of transactional resource:

- JTA transactional resources.
  A JTA transaction resource is a XA resource as defined by JTA.

- Non-JTA transactional resources.
  A non-JTA transactional resource supports transactions in a resource specific manner, such as a JDBC 2.0 driver that does not implement `javax.transaction.xa.XAResource`.

The SLEE specification does not require a SLEE to support JTA transactional resources.

If the SLEE implementation uses JTA to implement transactions and the resource implements JTA transactions, or if the resource supports transactions in a resource specific manner and the SLEE is integrated with this resource specific transaction API, when a method on the resource is invoked within a transaction the SLEE implementation may:

- Invoke the resource with no transaction context.
- Treat each invocation of the resource as a single transaction.
- Invoke the resource under the event handler transaction context.
- Combine multiple invocations to multiple resources under the calling transaction.

The exact characteristics depend on the implementation of both the SLEE and the resource.

A SLEE may integrate with EJB 2.0 containers to support transaction context propagation when methods on EJB interfaces are invoked within a transaction.

9.10 FIFO ordering of asynchronously fired events

When more than one event is fired on a particular Activity Context within the same transaction, the order of receipt of the events must be the order of the firing of events.

9.11 Non-transactional resources.

It is highly likely that resources used by SBBs within the SLEE are non-transactional. For example many call control resources such as soft switches, call agents, class 4 or 5 switches, SIP proxies, … etc. do not provide transactional semantics.

The SBB Developer must be aware that combining transactional and non-transactional resources in the same transaction may violate properties of the transactional programming model such as the atomicity guarantee. For example a transaction may update transactional state and then modify a non-transactional resource. If the transaction is rolled back, updates to the non-transactional resource remain.

9.12 Transactions exception handling and rolled back handling

There are two cases for the behavior of the SLEE when a method invocation throws an exception. These cases are:

- Non-SLEE originated method invocation.
- SLEE originated method invocation.

It can be seen from these two cases that the SLEE is the initiator in one case and SBB application logic is the initiator in the other. As SBBs are of arbitrary complexity, SBBs are expected to handle exceptional situations through a callee or caller/callee local interface contract. The SLEE is not of arbitrary complexity, its semantics are defined in this specification. Therefore if the SLEE initiates an invocation, a mechanism must be defined to allow the SLEE to inform the SBB of some unexpected situation (for example a resource fails after the SBB event handler invocation returns and before the transaction has committed causing the transaction to rollback, or a deadlock occurs causing the transaction to be rolled back).

Both SLEE originated and non-SLEE originated method invocations may also explicitly cause transactions to roll back through the use of the `setRollbackOnly` method regardless of whether the method invocation returns by throwing an exception or not. Resources may also request that transactions be rolled back.

Errors related to SLEE originated or non-SLEE originated method invocations may occur outside of invocation itself. Resource failures may cause these errors. These errors may also cause transactions to roll back.

9.12.1 Non-SLEE originated method invocation

If a non-SLEE originated method invocation returns by throwing a checked exception, then the following occurs:

- The state of the transaction is unaffected
- The checked exception is propagated to the caller.
  It is expected that the caller will have the appropriate logic to handle the exception.

If a non-SLEE originated method invocation returns by throwing a `RuntimeException`, then:

- The transaction is marked for rollback.
- The SBB object that was invoked is discarded, i.e. is moved to the Does Not Exist state.
- A `javax.slee.TransactionRolledBackLocalException` is propagated to the caller.

The transaction will eventually be rolled back when the highest level SLEE originated invocation returns as described in Section 9.12.2.

The `sbbRolledBack` method is not invoked for an SBB originated method transaction because the transaction is only marked for rollback and will only be rolled back when the highest level SLEE originated invocation returns. The `sbbRolledBack` method is only invoked on the SBB entity invoked by the SLEE as part of a SLEE originated invocation. (Clarified in 1.1)

If the `RuntimeException` propagates to the highest level (i.e. the SLEE originated method invocation returns by throwing a `RuntimeException`) the SLEE originated method invocation exception handling mechanism is initiated.

9.12.2 SLEE originated method invocation

If all the method invocations belonging to SLEE originated invocation return without throwing an exception or by throwing only checked exceptions, then the following occurs:

- If the transaction is not marked for rollback, then:
  - The transaction is moved to the Committing state, i.e. attempt to commit transaction.
  - If the transaction cannot be committed, then:
    - The transaction is rolled back.
    - If the SLEE originated method invocation has a target SBB entity (see Section 9.8.2), the SLEE eventually invokes the `sbbRolledBack` method in a new transaction on another SBB object that represents the same SBB entity as described in Section 6.10.1.
- If transaction is marked for rollback, then:
  - The transaction is rolled back.
  - If the SLEE originated method invocation has a target SBB entity (see Section 9.8.2), the SLEE eventually invokes the `sbbRolledBack` method in a new transaction on another SBB object that represents the same SBB entity as described in Section 6.10.1.

The only SLEE originated method invocations that may throw a checked exception are `sbbCreate` and `sbbPostCreate` method invocations. If an `sbbCreate` or an `sbbPostCreate` method does not want the SBB entity to be created it may either throw a `CreateException` or invoke the `setRollbackOnly` method to mark the transaction for rollback (and return without throwing an exception).

If a SLEE originated method invocation returns by throwing a `RuntimeException`, then the following occurs:

- The transaction is marked for rollback.
- The `sbbExceptionThrown` callback method is invoked as defined in Section 6.9.

- The SBB object that was invoked is discarded, i.e. moved to the Does Not Exist state.
- The transaction is eventually rolled back.
- If the SLEE originated method invocation has a target SBB entity (see Section 9.8.2), and the SLEE was not invoking the `sbbRolledBack` method on the target SBB entity the SLEE eventually invokes the `sbbRolledBack` method in a new transaction on another SBB object that represents the same SBB entity as described in Section 6.10.1. For more information on the invocation of `sbbRolledBack` and the transaction it runs within please see section 9.12.4.

If a transaction that includes a SLEE originated logical cascading removal method invocation does not commit (because the `sbbRemove` method of one of SBBs invoked returns by throwing a `RuntimeException` or marks the transaction for roll back, or for some other reasons like optimistic concurrency control conflicts, deadlocks, or internal SLEE problems), the SLEE is expected to forcefully remove the SBB entity tree that would have been removed by the SLEE originated logical cascading removal method invocation after the SLEE has invoked the `sbbRolledBack` callback corresponding to the transaction that did not commit.

9.12.3 SLEE originated transaction exception and rolled back handling example

The following example illustrates the behavior of SLEE when a `RuntimeException` propagates from a SLEE originated method invocation. The SLEE catches the exception, marks the transaction for rollback, invokes the `sbbExceptionThrown` method on the SBB object and rolls back the transaction. It then starts a new transaction and invokes the `sbbRolledBack` on another SBB object. The second SBB object has access to the same persistent state, i.e. it is assigned to the same SBB entity (see Section 6.2). It then commits the transaction. Note that the diagram assumes that the first SBB object is already in the ready state.

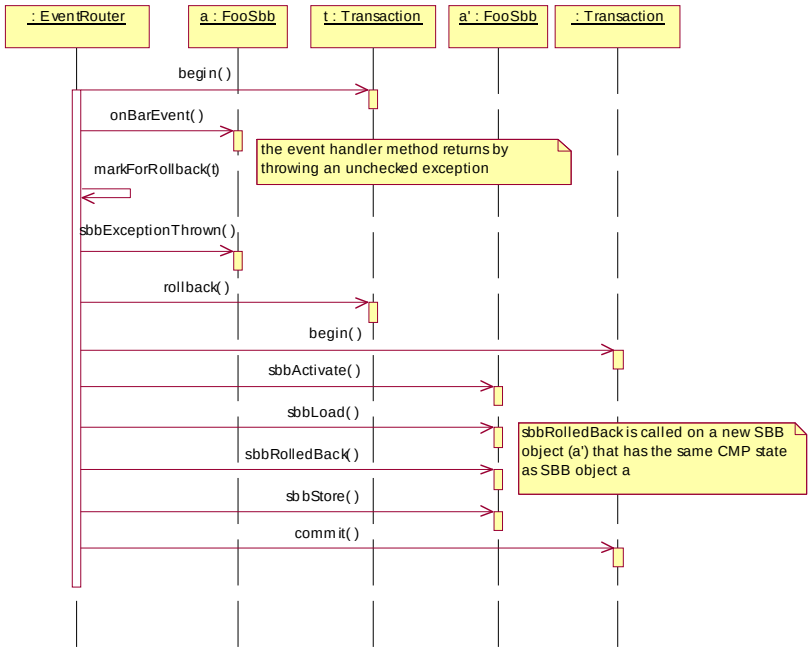

Figure 12 Transactional behavior of an SBB invocation that returns abnormally

9.12.4 Transaction rolled back callback handling

Changed in 1.1: Updated to indicate how the SLEE must perform rolled back callbacks when a transaction that contains multiple target SBB entities which received a SLEE originated invocation is rolled back.

Whenever a transaction rolls back, the SLEE must deliver a transaction rolled back callback for each SLEE originated invocation in the transaction that has a target SBB entity, and was not invoking the `sbbRolledBack` method[11]. The SLEE delivers each of these transactions rolled back callbacks by invoking the `sbbRolledBack` method in a transaction on another SBB object that represents the corresponding target SBB entity.

The transaction used to deliver the rolled back callbacks must be a new transaction. The new transaction must include the delivery of a rolled back callback to each target SBB entity which received a SLEE originated invocation which was not invoking the `sbbRolledBack` method in the transaction that rolled back.

For example if there is a transaction which delivered one event to three SBB entities and that transaction rolls back, then there will be one new transaction which will invoke the `sbbRolledBack` method on each of the three SBB entities that were invoked in the first transaction.

9.13 Local transaction enhancements

Added in 1.1.

Local transactions imply that the SLEE implementation is the only system executing with a single global transaction. The most common case of local transactions is where the SLEE implementation creates a new transaction in order to deliver an event that does not have an existing transaction context[12]. In general signaling protocols do not provide a transaction context.

The local transaction model allows multiple SBB entities that receive the same event to be invoked within the same transaction context. This enhancement enables the SLEE implementation to optimize delivery of events and gives the developer more flexibility when implementing a particular finite-state-machine. The local transaction model is automatic, i.e. there is no SBB visible transaction demarcation API.

9.14 Transactions and Resource Adaptors

Added in 1.1.

Resource adaptors may demarcate transaction boundaries. The interface used by resource adaptors for transactions is the Java Transaction API (JTA) version 1.0.1. The SLEE specification defines interfaces for performing asynchronous commit and rollback of transactions – these interfaces are specified as extensions of the Java Transaction API (JTA) 1.0.1 interfaces.

The use of JTA in SLEE has the following restrictions:

- Application components (SBBs, Profiles, and any classes which they include directly or indirectly) cannot demarcate transaction boundaries; the SLEE demarcates transaction boundaries on behalf of application components. Therefore any attempt to demarcate a transaction boundary by application code in SLEE is undefined.
- Resource adaptors do not use the `javax.transaction.UserTransaction` interface, as this interface is designed for use by certain J2EE application components, and client applications external to J2EE implementations. Therefore the use of `javax.transaction.UserTransaction` by resource adaptors in SLEE is undefined.
- Operations that reference `javax.transaction.XAResource` (that is, methods declared in the interface of `javax.transaction.XAResource`, or which specify a signature that in-

[11] If a transaction which only includes rolled back deliveries rolls back, then the rolled back deliveries do not get "redelivered" in a new transaction. Doing so would cause a loop.

[12] One example of this would be a SIP Resource Adaptor receiving a SIP INVITE message and then firing a SIP event to the SLEE for processing.

cludes a reference to a `javax.transaction.XAResource`) are unspecified in SLEE. *This may change in a future release of this specification.*

The rationale for restricting the use of JTA within SLEE is as follows:

- To adapt the JTA interface to the transaction boundary demarcation of the SLEE.
- The need for distributed transactions within SLEE has yet to be required by SLEE users and vendors. However, if at some future stage these requirements are evident it is possible that `XAResource` operations may be included in the SLEE.

9.14.1 `SleeTransaction` interface

The `SleeTransaction` interface defines a transaction in the SLEE. The interface allows operations to be performed against the target transaction. The SLEE is responsible for implementing SLEE transactions. The `SleeTransaction` interface extends the `javax.transaction.Transaction` interface.

The `SleeTransaction` interface is defined as follows:

```
package javax.slee.transaction;

import javax.transaction.Transaction;
import javax.transaction.RollbackException;
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.SystemException;
import javax.transaction.NotSupportedException;

public interface SleeTransaction extends javax.transaction.Transaction {
        public void asyncCommit(CommitListener commitListener)
                throws  IllegalStateException,
                        SecurityException;

        public void asyncRollback(RollbackListener rollbackListener)
                throws  IllegalStateException,
                        SecurityException;

        public boolean delistResource(XAResource xaRes, int flag)
                throws IllegalStateException,
                        SystemException;

        public boolean enlistResource(XAResource xaRes)
                throws RollbackException,
                        IllegalStateException,
                        SystemException;
        public boolean equals(Object other);
        public int hashCode();
        public String toString();
}
```

- The `asyncCommit` method.
  This method requests that the transaction represented by this `SleeTransaction` object asynchronously commit. The calling thread is not required to have the same transaction associated with the thread. This method initiates the commit operation and may return before the transaction completes.

  This method takes the following optional argument:

  - The `commitListener` argument is an object that implements the `javax.slee.transaction.CommitListener` interface. If this argument is `null` the SLEE will not attempt to invoke the methods defined on the `CommitListener` interface when the status of the commit operation is available, i.e. there is no way to know if the transaction is committed successfully or not. If the argument is non-`null` the SLEE will invoke the methods on defined on the `CommitListener` interface to provide the status of the commit operation when known.

This method may throw the following exceptions:

- `java.lang.IllegalStateException`.
  This exception is thrown by the target object if the object is not associated with a transaction.
- `java.lang.SecurityException`.
  This exception is thrown to indicate that the thread is not allowed to commit the transaction.

- The `asyncRollback` method.
  This method requests that the transaction represented by this `SleeTransaction` object asynchronously roll back. The calling thread is not required to have the same transaction associated with the thread. This method initiates the rollback operation and may return before the transaction completes.

  This method takes the following optional argument:

  - The `rollbackListener` argument is an object that implements the `javax.slee.transaction.RollbackListener` interface. If this argument is `null` the SLEE will not attempt to invoke the methods defined on the `RollbackListener` interface when the status of the rollback operation is available, i.e. there is no way to know if the rollback operation encountered any problems. If the argument is non-null the SLEE will invoke the methods on defined on the `RollbackListener` interface to provide the status of the rollback operation when known.

  This method may throw the following exceptions:

  - `java.lang.IllegalStateException`.
    This exception is thrown by the target object if the object is not associated with a transaction.
  - `java.lang.SecurityException`.
    This exception is thrown by the transaction if the current thread is not allowed to roll back the transaction.

- The `enlistResource` method.
  This method overrides the `enlistResource` method in the interface `javax.transaction.Transaction`. In SLEE this method is undefined.
- The `delistResource` method.
  This method overrides the `delistResource` method in the interface `javax.transaction.Transaction`. In SLEE this method is undefined.
- The `toString` method.
  This method overrides the `toString` method in `java.lang.Object`. It provides a String representation of a `SleeTransaction` object. The String representation of `SleeTransaction` objects are subject to equality requirements. That is if two `SleeTransaction` objects are equal their String representations must be equal; conversely if two `SleeTransaction` objects are not equal their String representations must not be equal.

9.14.1.1 `SleeTransaction` Equality and Hash Code

The SLEE must implement the `SleeTransaction` object's `equals` method to allow comparison between the target object and another `SleeTransaction` object. The `equals` method should return `true` if the target object and the parameter object both refer to the same global transaction.

The following provides an example of how the SLEE or an Resource Adaptor may perform a comparison between two transaction objects.

```
SleeTransaction tx = SleeTransactionManager.getSleeTransaction();
SleeTransaction otherTx = ..
```

```
boolean isSame = tx.equals( otherTx );
```

In addition, the SLEE must implement the `SleeTransaction` object's `hashCode` method so that if two `SleeTransaction` objects are equal, they have the same hash code. However, the converse is not necessarily true. Two `SleeTransaction` objects with the same hash code are not necessarily equal.

9.14.2 `SleeTransactionManager` interface

The `SleeTransactionManager` interface defines the methods that allow resource adaptor objects to manage transaction boundaries. The SLEE is responsible for implementing the SLEE transaction manager. The SLEE transaction manager is not required to support nested transactions. The `SleeTransaction-Manager` interface extends the `javax.transaction.TransactionManager` interface.

The `SleeTransactionManager` interface is defined as follows:

```
package javax.slee.transaction;

import javax.transaction.Transaction;
import javax.transaction.TransactionManager;
import javax.transaction.RollbackException;
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.SystemException;
import javax.transaction.NotSupportedException;

public interface SleeTransactionManager extends javax.transaction.TransactionManager{
        public SleeTransaction beginSleeTransaction()
                throws  NotSupportedException,
                        SystemException;
        public SleeTransaction getSleeTransaction()
                throws  SystemException;
        public void asyncCommit(CommitListener commitListener)
                throws  IllegalStateException,
                        SecurityException;
        public void asyncRollback(RollbackListener rollbackListener)
                throws  IllegalStateException,
                        SecurityException,
                        SystemException;
        public SleeTransaction asSleeTransaction(Transaction transaction)
                throws  IllegalArgumentException,
                        NullPointerException,
                        SystemException;
}
```

- The `beginSleeTransaction` method.
  This method starts a new transaction and associates the transaction context with the calling thread. A `SleeTransaction` object representing the new transaction is the return value of this method.
  This method may throw the following exceptions:
  - `javax.transaction.NotSupportedException`.
    This exception is thrown if the calling thread is already associated with a transaction and the transaction manager does not support nested transactions.
  - `javax.transaction.SystemException`.
    This exception is thrown by the transaction manager to indicate that it has encountered an unexpected error condition.
- The `getSleeTransaction` method.
  This method returns the SleeTransaction object that represents the transaction context of the calling thread. If there is no transaction context associated with the calling thread then this method returns `null`.
  This method may throw the following exceptions

  - `javax.transaction.SystemException`.
    This exception is thrown by the transaction manager to indicate that it has encountered an unexpected error condition.

- The `asyncCommit` method.
  This method requests the initiation of the commit of the transaction associated with the current thread. Upon completion of this method the thread is no longer associated with a transaction[13]. This method differs from the inherited `commit` method in that control may be returned to the caller before the transaction has committed or rolled back.

  This method takes the following optional argument:

  - The `commitListener` argument is an object that implements the `javax.slee.transaction.CommitListener` interface. If this argument is `null` the SLEE will not attempt to invoke the methods defined on the `CommitListener` interface when the status of the commit operation is available, i.e. there is no way to know if the transaction is committed successfully or not. If the argument is non-null the SLEE will invoke the methods on defined on the `CommitListener` interface to provide the status of the commit operation when known.

  This method may throw the following exceptions:

  - `java.lang.IllegalStateException`.

    This exception is thrown to indicate that the thread is not associated with a transaction.

  - `java.lang.SecurityException`.
    This exception is thrown to indicate that the thread is not allowed to commit the transaction.

- The `asyncRollback` method.
  This method requests the initiation of the rollback of the transaction associated with the current thread. Upon completion of this method the thread is no longer associated with a transaction[14].This method differs from the inherited `rollback` method in that control may be returned to the caller before the transaction has rolled back.

  This method takes the following optional argument:

  - The `rollbackListener` argument is an object that implements the `javax.slee.transaction.RollbackListener` interface. If this argument is `null` the SLEE will not attempt to invoke the methods defined on the `RollbackListener` interface when the status of the rollback operation is available, i.e. there is no way to know if the rollback operation encountered any problems. If the argument is non-null the SLEE will invoke the methods on defined on the `RollbackListener` interface to provide the status of the rollback operation when known.

  This method may throw the following exceptions:

  - `java.lang.IllegalStateException`.
    This exception is thrown by the transaction manager if the current thread is not associated with a transaction.

  - `java.lang.SecurityException`.
    This exception is thrown to indicate that the thread is not allowed to roll back the transaction.

---

[13] If the current thread has an associated transaction, then calling `SleeTransactionManager.asyncCommit()` is equivalent to calling `SleeTransactionManager.getSleeTransaction().asyncCommit()`

[14] If the current thread has an associated transaction, then calling `SleeTransactionManager.asyncRollback()` is equivalent to calling `SleeTransactionManager.getSleeTransaction().asyncRollback()`

- The `asSleeTransaction` method.
  This method narrows a `javax.transaction.Transaction` object to a `javax.slee.transaction.SleeTransaction` object. The narrowed object is the return value.
  This method takes the following argument:
  - The `javax.transaction.Transaction` argument represents a transaction started by this transaction manager.

  This method may throw the following exceptions:
  - `java.lang.IllegalArgumentException`.
    This exception is thrown if the argument does not represent a transaction started by this transaction manager.
  - `java.lang.NullPointerException`.
    This exception is thrown if the argument is `null`.
  - `javax.transaction.SystemException`.
    This exception is thrown if the transaction manager encounters an unexpected error condition.

9.14.3 `CommitListener` Interface

The `CommitListener` interface defines the callback operations that enable a Resource Adaptor to be notified of the outcome of an asynchronous commit operation. An asynchronous commit operation is performed by invoking the `asyncCommit` method on either the `SleeTransaction` or the `SleeTransactionManager` interfaces.

The `CommitListener` interface is defined as follows:

```
package javax.slee.transaction;

import javax.transaction.RollbackException;
import javax.transaction.HeuristicMixedException;
import javax.transaction.HeuristicRollbackException;
import javax.transaction.SystemException;

public interface CommitListener{
       public void committed();
       public void rolledBack(RollackException rbe);
       public void heuristicMixed(HeuristicMixedException hme);
       public void heuristicRollback(HeuristicRollbackException rbe);
       public void systemException(SystemException se);
}
```

All methods on this interface are invoked by the SLEE, and are implemented by the Resource Adaptor Developer. Implementations of these methods are expected to be re-entrant safe, and multi-thread safe. The runtime environment for these methods[15] is specified in Chapter 12.

The methods on this interface are defined as follows:

- The `committed` method.
  This method is invoked by the SLEE to indicate the transaction commit was successful.
- The `rolledBack` method.
  This method is invoked by the SLEE to indicate the transaction did not commit rather it rolled back.

  This method takes the following argument:

[15] The runtime envirironment defines various properties – for example the associated security permissions.

  - The `rbe` argument is a `RollBackException` object which may provide information about the cause of the roll back.

- The `heuristicMixed` method.
  This method is invoked by the SLEE to report that a heuristic decision was made and that some relevant updates have been committed while others have been rolled back.

  This method takes the following argument:

  - The `hme` argument is a `HeuristicMixedException` object, which may provide information about the heuristic decision.

- The `heuristicRollback` method.
  This method is invoked by the SLEE to indicate that a heuristic decision was made and that all relevant updates have been rolled back.

  This method takes the following argument:

  - The `hre` argument is a `HeuristicRollbackException` object, which may provide information about the heuristic decision.

- The `systemException` method.
  This method is invoked by the SLEE to indicate that the transaction manager encountered an unexpected error.

  This method takes the following argument:

  - The `se` argument is a `SystemException` object, which may provide information related to the unexpected error.

9.14.4 `RollbackListener` Interface

The `RollbackListener` interface defines the callback operations that enable a Resource Adaptor to be notified of the outcome of an asynchronous roll back operation. An asynchronous roll back operation is performed by invoking the `asyncRollback` method on either the `SleeTransaction` or the `SleeTransactionManager` interfaces.

The `RollbackListener` interface is defined as follows:

```
package javax.slee.transaction;

import javax.transaction.SystemException;

public interface RollbackListener{
       public void rolledBack();
       public void systemException(SystemException se);
}
```

All methods on this interface are invoked by the SLEE, and are implemented by the Resource Adaptor Developer. Implementations of these methods are expected to be re-entrant safe, and multi-thread safe. The runtime environment for these methods[16] is in Chapter 12.

The methods on this interface are defined as follows:

- The `rolledBack` method.
  This method is invoked by the SLEE to indicate the transaction rolled back successfully.

- The `systemException` method.
  This method is invoked by the SLEE to indicate that the transaction manager encountered an unexpected error.

  This method takes the following argument:

[16] The runtime envirironment defines various properties – for example the associated security permissions.

- The `se` argument is a `SystemException` object, which may provide information related to the unexpected error.

9.14.5 Transaction rolled back callback handling

When a transaction rolls back, the SLEE should deliver a transaction rolled back callback for each SLEE originated invocation in the transaction that has a target SBB entity. The SLEE delivers each of these transaction rolled back callbacks by invoking the `sbbRolledBack` method in a new transaction on another SBB object that represents the corresponding target SBB entity.

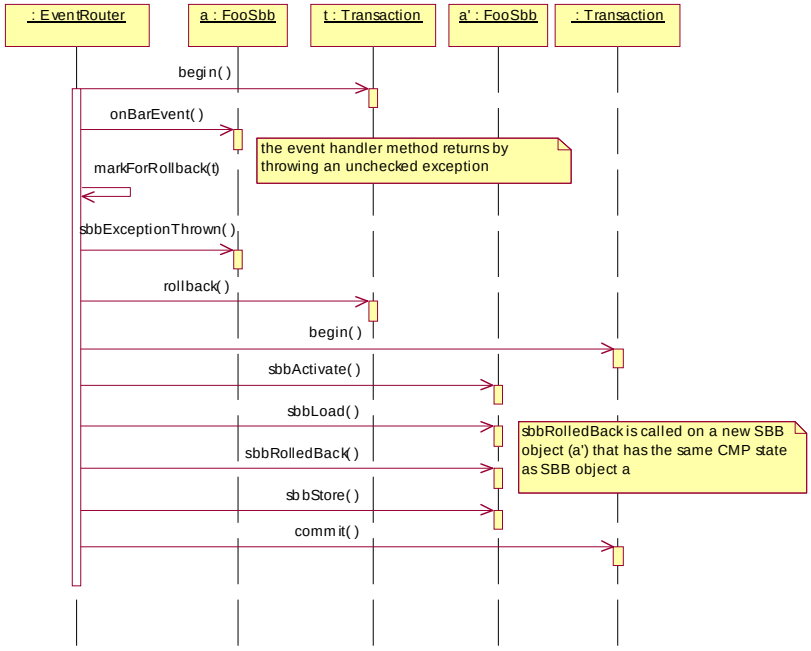

Figure 13 Transactional behavior of an SBB invocation that returns abnormally

Chapter 10 Profiles and Profile Specifications

Changed in 1.1: The 1.1 Profile contract significantly improves upon the 1.0 Profile contract. The most significant changes include allowing components, such as SBBs or Resource Adaptors, to update Profiles and allowing the Profile Specification to include business methods, hide methods/attributes, … etc. Because of significant changes to this chapter, changes introduced by the 1.1 specification are not individually annotated in this chapter.

A Profile contains data and may contain associated logic. Examples include subscriber data such as a call forwarding number or service data, such as a SIP registration records. The schema of the Profile defines the attributes that may be provisioned in the Profile. A Profile Table contains zero or more Profiles that conform to the same schema. A Profile Specification defines the interfaces, classes, and deployment descriptor elements needed to define a Profile schema and the interfaces used to provision and access a Profile that conforms to the Profile schema. Zero or more Profile Tables can have the same Profile Specification.

10.1 Compatibility with 1.0

A 1.1 SLEE implementation must implement and support both the 1.0 and 1.1 Profiles and Profile Specification contracts. The SLEE must determine which version of the contract applies to a particular Profile Specification based on the deployment descriptor for the Profile Specification. The SLEE identifies the version using the XML document type identified in the deployment descriptor.

In addition, only 1.1 components can reference and use 1.1 Profile Specifications.

10.2 Introduction

A Profile Specification Developer defines a Profile Specification. Each Profile Specification specifies a kind of provisioned data, activity-related state, configuration information, or other information required by an SBB or Resource Adaptor to perform its function. The Profile Specification describes the schema of Profile Tables created from the Profile Specification by the Administrator. More precisely, it defines the set of attributes of every Profile Table created from the Profile Specification. Every Profile in these Profile Tables has the same attributes.

The Profile Specification Developer defines a Profile Specification through a set of Java interfaces, classes, and a deployment descriptor. These interfaces, classes, and deployment descriptor define the three views of a Profile. The three views are the SLEE Component object view (see Section 10.5.1), the Profile Specification Developer's management view (see Section 10.5.2), and the Administrator's view (see Section 10.5.3).

The Profile Specification Developer packages these Java class files and the deployment descriptor into a Profile Specification jar file. The Service Deployer or Resource Adaptor Deployer packages the Profile Specification into a deployable unit. The Administrator deploys this deployable unit into the SLEE. Once the Profile Specification is deployed, the Administrator can create Profile Tables of the Profile Specification. Each Profile Table has a name, known as the Profile Table name. Once a Profile Table has been created, the Administrator may add Profiles to the Profile Table. Each Profile within a Profile Table must have a unique name within the Profile Table. This name is known as the Profile name. The Profile Table name and Profile name together uniquely identify a Profile within the SLEE. This is known as the Profile identifier.

SLEE components (SBBs and Resource Adaptors) and management clients can have write access to Profiles (SBBs had a read-only view of Profiles in the SLEE v1.0 specification). Profile updates by SLEE components enable applications to modify Profile data during service execution.

Note: It is common practice that the data stored in Profiles within the SLEE is a subset of the data within the master provisioning system (within the OSS backend). If there is a need to synchronize the data in the SLEE and the master provisioning system there are several options:

- *The application may take responsibility for resynchronization - for example Profiles in the SLEE and records in the provisioning system could be timestamped and an application can be used to synchronize changes between the SLEE Profiles and the master provisioning system based on*

timestamps. This application could be realized externally from the SLEE (via the SLEE Profile Provisioning interfaces) or internally to the SLEE (via SBB or Resource Adaptor code and the SLEE Component object view APIs).

- *The SLEE implementation may communicate directly with the master provisioning system in which case the responsibility for synchronization is taken by the SLEE implementation. In this case the semantics for the communication are the responsibility of the SLEE vendor.*

10.2.1 Changes made during JAIN SLEE 1.1

In SLEE 1.0 the SBB Developer had a read-only view of Profiles whilst management clients had a read-write view. In SLEE 1.1 both SBBs and Resource Adaptors may have a read-write view of Profiles and management clients retain their read-write view.

The following table provides a summary of the most important additions and/or modifications made to the Profile chapter during SLEE1.1.

Profile abstract class	In order to provide SLEE components with a read-write view of Profiles and to allow arbitrary application logic to be associated with Profile data, a new Profile class was required. This class is the Profile abstract class. The Profile abstract class replaces the Profile Management abstract class from SLEE 1.0. SLEE 1.1 implementations are required to support the 1.0 Profile Management abstract class when a 1.0 Profile Specification DTD is referenced for backwards compatibility and the 1.1 Profile abstract class when a 1.1 Profile Specification DTD is referenced.
Profile Local interface	This is a new interface defined in SLEE 1.1. The Profile Local interface is used by SLEE components to invoke business methods on a Profile. The interface can be used so that Profiles can store data and execute program logic associated with that data.
Profile Table	This is a new interface defined in SLEE 1.1. The Profile Table interface is defined so that SLEE components may create, query, and remove Profiles within a Profile Table.
Profile Usage Parameters Interface	The SLEE 1.1 specification allows a Profile Specification to define a Usage Parameters Interface to collect usage information about Profiles in a Profile Table.
Permitted Combinations and associated views of a Profile Specification	Due to the support for SLEE component read-write access to Profiles, the different component views of Profiles and their associated interfaces have been updated.

Profile Query Operations	In order to provide flexible query operations for Profiles the SLEE 1.1 specification defines two types of queries - static and dynamic. Static queries are declared prior to the deployment of a Profile Specification into the SLEE. Dynamic queries may be declared after a Profile Specification has been deployed.
Resource Info Profile Specification	This has been deprecated in SLEE 1.1. It is no longer necessary as Resource Adaptors have access to Profiles. This Profile Specification may be removed in a future version of the SLEE specification.
Profile Specification	There are two definitions of the Profile Specification, one for SLEE 1.0 and one for SLEE 1.1. This is due to the changes in the Profile contract between 1.0 and 1.1. SLEE 1.1 implementations are required to support both the 1.0 and 1.1 Profile Specifications.
Profile Specification Developer's SLEE Component object view	The SBB Developer's SBB object view has been renamed to the Profile Specification Developer's SLEE Component object view as both SBBs and Resource Adaptors may view Profiles. SBBs and Resource Adaptors both share the same view – the SLEE Component object view.

10.2.2 Relationships

Profile Specifications, Profile Tables, and Profiles participate in the following relationships:

- Profile Specifications and Profile Tables: 1-to-many
  The Administrator may create multiple Profile Tables from the same Profile Specification. However, each Profile Table is created to conform to exactly one Profile Specification.
- Profile Tables and Profiles: 1-to-many
  Each Profile may belong to only one Profile Table. A Profile Table contains a default Profile (see 10.2.6) and zero or more user-defined Profiles.
- Profiles and Profiles: many-to-many
  Each Profile may reference another Profile or itself (see Section 10.6.1). The SLEE does not impose any restrictions on the Profile references.
- Address Profile Tables and Services: 1-to-many
  Each Service may optionally reference an Address Profile Table. The same Address Profile Table may be referenced by more than one Service (see Section 10.24.1).

10.2.3 META-Model

Figure 14 provides a UML META-Model of Profiles.

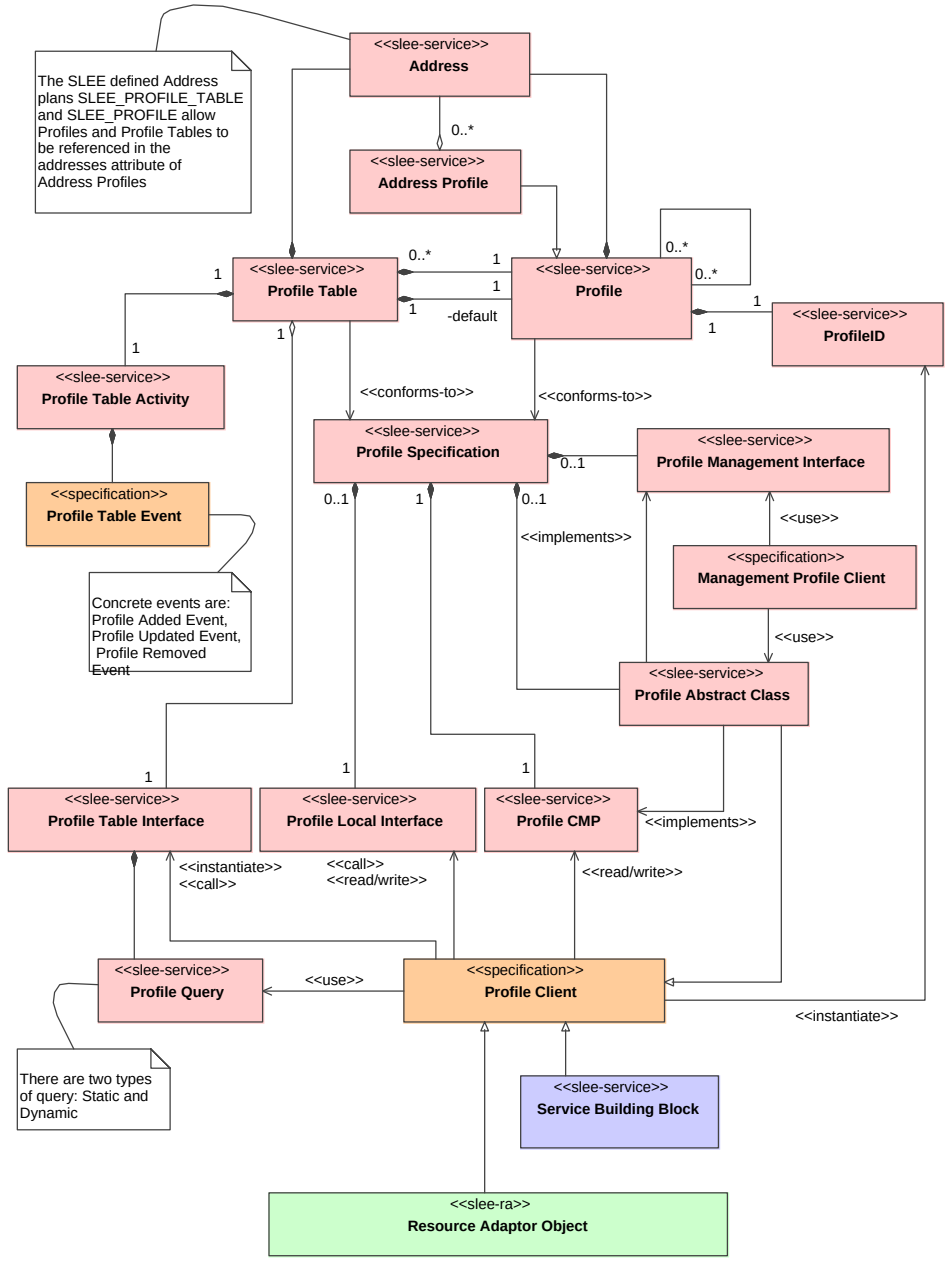

Figure 14 UML Meta-Model of Profiles

10.2.4 Profile Table names, Profile names, and Profile identifiers

The following names and identifiers are used to identify Profile Tables and Profiles.

- Profile Table name.
  The Administrator must assign a globally unique name to each Profile Table, known as the Profile Table name. The Java type for Profile Table names is `java.lang.String`. A Profile Table name cannot be `null` or zero-length, and may only contain letter or digit characters as defined by `java.lang.Character.isLetterOrDigit`. Additionally, any other character in the

Unicode range 0x0020-0x007e may be included in a Profile Table name, except for the '/' character (Unicode 0x002f).

- Profile name.
  Each Profile within a Profile Table must have a unique name within the Profile Table. This name is known as the Profile name. The Java type for Profile names is `java.lang.String`. A Profile name cannot be `null` or zero-length, and may only contain letter or digit characters as defined by `java.lang.Character.isLetterOrDigit`. Additionally, any other character in the Unicode range 0x0020-0x007e may be included in a Profile name.
- Profile identifier.
  The Profile Table name and Profile name together uniquely identify a Profile within the SLEE. This is known as the Profile identifier. The Java type that encapsulates Profile identifiers is the `ProfileID` class (see Section 10.3).

10.2.5 Profile Table and Profile addresses

The following kinds of addresses also identify Profile Tables and Profiles within Profile Tables.

- Profile Table address.
  A Profile Table can also be identified by an `Address` object. The address plan of this `Address` object is `SLEE_PROFILE_TABLE` and the address string contains the Profile Table's name.
- Profile address.
  A Profile can also be identified by an `Address` object. The address plan of this `Address` object is `SLEE_PROFILE` and the address string contains the Profile identifier encoded as "profile_table_name/profile_name".

The address form of identifying Profile Tables and Profiles allows Profile Tables and Profiles to be identified in the `address` attribute of Address Profile Tables (see Section 10.24.1), and allows Profile Added Events, Profile Removed Events, and Profile Updated Events to be initial events.

10.2.6 Default Profile of a Profile Table

Each Profile Table has a *default* Profile that is specific to the Profile Table.

- When the Profile Table is created, the persistent state of the default Profile is initialized by the `profileInitialize` method (see Section 10.13.1.3) of the Profile abstract class (if one is defined).
- An Administrator can update the default Profile of the Profile Table through the SLEE's management interface.
- When an Administrator creates a new Profile in the Profile Table, the SLEE initializes the persistent state of the new Profile with the values contained in the default Profile. The default Profile of a Profile Table provides the initial values for the attributes of new Profiles of the same Profile Table.

10.3 `ProfileID` class

The public interface of the `ProfileID` class is as follows:

```
package javax.slee.profile;

import java.io.Serializable;
import javax.slee.Address;

public class ProfileID implements Serializable {
       public ProfileID(String profileTableName, String profileName)
              throws NullPointerException, IllegalArgumentException { … }
       public ProfileID(Address address)
              throws NullPointerException, IllegalArgumentException { … }
       public String getProfileTableName() { … }
```

```
        public String getProfileName() { … }
        public void setProfileID(String profileTableName, String profileName)
               throws NullPointerException, IllegalArgumentException { … }
        public String toString() { … }
        public Address toAddress() { … }
        public int hashCode() { … }
        public boolean equals(Object obj) { … }
}
```

- The two-argument constructor.
  This constructor creates a `ProfileID` object identifying the Profile specified by the `profileTableName` and `profileName` arguments.
- The one-argument constructor.
  This constructor creates a `ProfileID` object identifying the Profile specified by the `address` argument. The Address Plan of the `address` argument must be `AddressPlan.SLEE_PROFILE`.
- The `getProfileTableName` and `getProfileName` methods.
  These method returns the Profile Table name and Profile name of the Profile identified by this Profile identifier respectively.
- The `setProfileID` method.
  This method allows the Profile identified by this Profile identifier to be changed. The `profileTableName` and `profileName` arguments identify the new Profile for the Profile identifier.
- The `toString` method.
  This method returns a string representation of the Profile identifier.
- The `toAddress` method.
  This method returns an `Address` object that identifies the same profile as this Profile identifier.
- The `equals` method.
  This method indicates whether the specified object (specified by the `obj` argument) is equal to this Profile identifier. The specified object is equal to this Profile identifier if the specified object is a Profile identifier and the names of the Profile Table and Profile in the specified Profile identifier is the same as this Profile identifier.
- The `hashCode` method.
  The implementation of this method should provide a reasonable hash code value for the Profile identifier.

The above methods may throw the following exceptions:

- `java.lang.NullPointerException`.
  If any of a method's arguments (`profileTableName`, `profileName`, or `address`) are `null`, the method throws a `NullPointerException`.
- `java.lang.IllegalArgumentException`.
  If the `profileTableName` argument contains the "/" character, or the `address` argument does not have an address plan of `SLEE_PROFILE` or the address string is not valid for the `SLEE_PROFILE` address plan, the method throws an `IllegalArgumentException`.

10.4 Profile Specification

Each Profile Specification includes:

- A Profile CMP interface.
  This interface defines the schema of the Profile Table, i.e. it defines the attributes of each Profile in the Profile Table. The Profile Specification Developer declares each attribute using a pair of get and set accessor methods in this interface.

- A Profile Local interface, optional.
  This interface declares the methods that are made available to internal SLEE components, e.g. SBBs and Resource Adaptors. These methods may include methods declared in the Profile CMP interface and additional business methods that interact and manipulate the attributes of a Profile that conforms to this Profile Specification.
  If a Profile Specification does not define a Profile Local interface, the SLEE uses an anonymous interface that extends the Profile CMP interface and the SLEE-defined `javax.slee.profile.ProfileLocalObject` interface as the Profile Local interface for the Profile Specification. This means that, in this case, Profile Local objects obtained for the Profile Specification may be safely typecast by SLEE Components to the Profile CMP interface of the Profile Specification.
- A Profile Management interface, optional.
  This interface declares the methods that are made available to management clients. These methods may include methods declared in the Profile CMP interface and additional Profile Specification Developer declared management methods (not methods defined in the SLEE specification defined `javax.slee.profile.Profile` interface) that interact and manipulate the attributes of a Profile that conforms to this Profile Specification.
- A Profile abstract class, optional.
  This abstract class implements the Profile CMP interface, the Profile Management interface (if defined by the Profile Specification), and the `javax.slee.profile.Profile` interface. The Profile Local interface is not directly implemented by the Profile abstract class, however the business methods defined in the Profile Local interface must be implemented in this abstract class. The Profile Specification Developer also implements the management methods declared in the Profile Management interface and the Profile life cycle methods declared in the `Profile` interface in this abstract class. The SLEE implements the get and set accessor methods of the Profile CMP interface in one or more derived classes of the abstract class when the Profile Specification is deployed. The methods implemented by the Profile Specification Developer can invoke the accessor methods defined in the Profile CMP interface to manipulate the attributes of a Profile that conforms to this Profile Specification. *In the SLEE 1.1 specification, the Profile abstract class replaces the Profile Management abstract class defined by SLEE 1.0.*
- A Profile Table interface, optional.
  This interface declares the methods that SLEE components use to interact with a Profile Table. It provides the ability to create, find, remove, and execute queries on Profiles. The Profile Table interface must extend the SLEE defined `javax.slee.profile.ProfileTable` interface. If a Profile Table interface is not specified for a Profile Specification, the SLEE defined `ProfileTable` interface is used by default.
- A Profile Specification deployment descriptor element.
  This element contains sub-elements that identify the above interfaces and abstract class of the Profile Specification. Attributes indexing directives and hints and Profile static queries are also defined in the deployment descriptor.

10.5 Views of a Profile Specification

There are three ways to view a Profile Specification.

- The SLEE Component object view.
  In this view, the Profile Specification Developer is concerned with the interfaces presented to objects such as SBB objects and/or Resource Adaptor objects to access data in a Profile conforming to the Profile Specification.
- The Profile Specification Developer's management view.
  In this view, the Profile Specification Developer is concerned with the interfaces and classes provided to and used by the SLEE management system to provision a Profile conforming to the Profile Specification.

- Administrator's view.
  In this view, the Administrator is concerned with the MBean interface derived by the SLEE from the Profile Specification when the Profile Specification is deployed.

10.5.1 SLEE Component object view

The SLEE Component object view replaces the SLEE 1.0 SBB Developer's SBB object view. The SLEE Component object view is defined by the Profile Local interface (see Section 10.7), the Profile Table interface (see Section 10.8), and the Profile abstract class (see Section 10.11). The Profile Local interface defines the methods that are made visible to SLEE Components and the Profile abstract class provides the implementations of these methods. The Profile Table interface defines methods that allow SLEE Components to create, find, remove, and execute queries on Profiles. A Profile object is an object instantiated by the SLEE at runtime that interacts with the persistent state of a Profile on behalf of a SLEE Component.

Note: SLEE 1.1 Profile abstract class differs from the SLEE 1.0 Profile Management abstract class in that the SLEE 1.1 Profile abstract class implements both the SLEE Component object view and the management view, while the SLEE 1.0 Profile Management abstract class only implements the management view. They also have different SLEE defined callback methods.

A SLEE 1.1 implementation must support both the SLEE 1.0 SBB Developer's SBB object view and the SLEE 1.1 SLEE Component object view of a Profile Specification.

- If the Profile Specification defines a Profile Local interface, then SLEE Components access a Profile through methods defined by the Profile Local interface.
- If the Profile Specification does not define a Profile Local interface, then SLEE Components access a Profile through methods defined by the Profile CMP interface, i.e. all get and set accessor methods declared in the Profile CMP interface are made visible to SLEE Components. This interface must comply with both the requirements of a Profile CMP interface and a Profile Local interface and the SLEE must allow Profile Local Objects of the Profile Specification to be Java typecast to the Profile CMP interface of Profile Specification.

Note: Profile modifications by SLEE components are not intended to be used to manage the Profile subsystem. Managing the Profile subsystem should be performed by an Administrator through the SLEE's management interfaces.

10.5.1.1 Profile read-only attribute

The SLEE Component object view provides both a read-only and read-write view for SLEE Components.

A `profile-read-only` boolean attribute defined in the `profile-spec` element determines whether the SLEE Component object view is read-only or read-write. (It does not affect the management view that continues to be read-write.) The default value of this attribute is "True". If this attribute is set to "True" then SLEE components may not create, remove or modify Profiles created from the Profile Specification. If this value is set to "False" SLEE Components are permitted to create, remove, or modify Profiles created from the Profile Specification.

The following behavior must be observed if the SLEE Component object view of a Profile Specification is read-only:

- If a SLEE Component attempts to modify the value of a Profile CMP field of a Profile created from the Profile Specification, for example by invoking a CMP setter method on the Profile CMP interface, by invoking a CMP setter method on the Profile Local interface, or by invoking a business method on the Profile Local interface that attempts to set the value of a Profile CMP field, the CMP field setter method in the Profile object throws a `javax.slee.profile.ReadOnlyProfileException`. In the case of a method on the Profile Local interface being invoked, the `ReadOnlyProfileException` thrown from the CMP setter method in the Profile object causes the current transaction to be marked for rollback if the `ReadOnlyProfileException` propagates unhandled out of the Profile object (see 10.7.5). In the case of a setter method on the Profile CMP interface being invoked, the

`javax.slee.profile.ReadOnlyProfileException` thrown from the CMP setter method in the Profile object is translated into a `java.lang.UnsupportedOperationException` for backwards compatibility with SLEE 1.0. Note that the transaction is not marked for rollback in this case.

- If a SLEE Component attempts to create or remove a Profile in a Profile Table created from the Profile Specification, the `create` or `remove` method invoked on the Profile Table interface throws a `javax.slee.profile.ReadOnlyProfileException`.
- If a SLEE Component attempts to remove a Profile via the `remove` method on the `javax.slee.profile.ProfileLocalObject` then the `remove` method throws a `javax.slee.profile.ReadOnlyProfileException.`

10.5.1.2 SLEE Component view combinations

The following table illustrates the allowed combinations for the Profile Specification Developer's SLEE Component view, and the resulting methods that are made available in the SLEE Component view.

Combinations	Profile CMP interface	Profile Local interface	Profile abstract class	Comments
1	X			All Profile CMP interface accessor methods are available to SLEE Components.
2	X	X		Only Profile CMP interface accessor methods that are also defined in the Profile Local interface are available to SLEE Components.
3	X		X	All Profile CMP interface accessor methods are available to SLEE Components. The Profile abstract class implements the life cycle callback methods.
4	X	X	X	Only methods defined in the Profile Local interface are available to SLEE Components. These methods may include Profile CMP accessor methods and business methods. The Profile abstract class implements the business methods and the life cycle callback methods.

Note: A method implemented in the Profile abstract class can be defined in both the Profile Local interface (e.g. as a business method) and the Profile Management interface (e.g. as a management method).

10.5.2 Profile Specification Developer's management view

The Profile Specification Developer's management view of a Profile Specification defines the Profile Specification specific methods that are made visible to management clients (see Section 10.5.3 for the Administrator's view).

10.5.2.1 Profile Specification Developer's management view

SLEE 1.1 defines the Profile Specification Developer's management view to include a Profile Management interface (see Section 10.10) and Profile abstract class (see Section 10.11). The Profile Management interface defines the methods that are made visible to management clients and the Profile abstract class provides the implementations of these methods. A Profile object is an object instantiated by the SLEE at runtime that interacts with the persistent state of a Profile on behalf the management client.

Note: SLEE 1.1 Profile abstract class differs from the SLEE 1.0 Profile Management abstract class in that the SLEE 1.1 Profile abstract class implements both the management view and the SLEE Component object view, i.e. the methods defined by the Profile Local interface, while the SLEE 1.0 Profile Management abstract class only implements the management view. They also have different SLEE defined callback methods.

- If the Profile Specification defines a separate Profile Management interface (i.e. this interface is not the same as the Profile CMP interface), then the management clients access a Profile through methods defined by the Profile Management interface.
- If the Profile Specification's Profile Management interface and Profile CMP interface reference the same class name, or the Profile Specification does not identify a Profile Management interface, then the management clients access a Profile through methods defined by the Profile CMP interface, i.e. all get and set accessor methods declared in the Profile CMP interface are made visible to management clients. This interface must comply with both the requirements of a Profile CMP interface and a Profile Management interface.

10.5.2.2 Profile Specification Developer's management view combinations

The following table illustrates the allowed combinations for the Profile Specification Developer's management view, and the resulting methods that are made available in the management view.

Combinations	Profile CMP interface	Profile Management interface	Profile abstract class	Comments
1	X			All Profile CMP interface accessor methods are available to management clients.
2	X	X		Only Profile CMP interface accessor methods that are also defined in the Profile Management interface are available to management clients.
3	X		X	All Profile CMP interface accessor methods are available to management clients. The Profile abstract class implements the life cycle callback methods.
4	X	X	X	Only methods defined in the Profile Management interface are available to management clients. These methods may include Profile CMP accessor methods and management methods. The Profile abstract class implements the man-

				agement methods and the life cycle callback methods.

Note: A method implemented in the Profile abstract class can be defined in both the Profile Local interface (e.g. as a business method) and the Profile Management interface (e.g. as a management method).

10.5.3 Administrator's view

The Administrator's view of a Profile Specification is the Profile Specification's JMX MBean interface, known as the Profile MBean interface (see Section 10.26). The Profile MBean interface is the external management interface that is visible to external management clients that interact with the Profiles specified by the Profile Specification.

The Profile MBean interface is generated by the SLEE when the Profile Specification is deployed from the methods defined in the Profile Management interface, if a Profile Management interface is defined, or from the Profile CMP interface if a Profile Management interface is not defined.

10.6 Profile CMP interface

The Profile CMP interface defines the collection of persistent attributes or CMP fields of the Profile Specification. All persistent attributes that are contained in a Profile must be included in the Profile CMP interface. At deployment time the SLEE implements the Profile CMP interface to provide the mechanism to support the actual persistence of the data values.

The following are the requirements for the Profile CMP interface:

- The Profile CMP interface must be defined in a named package, i.e. the class must have a `package` declaration.
- The Profile CMP interface must be declared as `public`.
- CMP fields are defined in the Profile CMP interface by the use of abstract get and set accessor methods with the following design pattern:

```
public abstract <attribute type> get<attribute name>();
public abstract void set<attribute name>(<attribute type> value);
```

  - Attribute names must be valid Java identifiers. The first letter of the attribute name in a get and set accessor method must be capitalized, as determined by `java.lang.Character.isUpperCase`. This means that `getFoo` is legal, while `getfoo` is not.
  - When a Profile CMP field is referenced in a Profile Specification deployment descriptor the name of the field must be a valid Java identifier and must begin with a lowercase letter, as determined by `java.lang.Character.isLowerCase`.
  - Each attribute must have both a get and set accessor method defined, and the corresponding types used in the get and set accessor methods must be the same.
  - The Java types of the Profile CMP fields are restricted to Java primitive types and Java serializable types, and arrays of such types. Furthermore, attribute types must conform to the requirements specified in Section 10.17.
  - CMP fields that hold dependent value classes.
    A dependent value class is a concrete class. A dependent value class may be a legacy class that the Profile Developer wishes to use internally within a Profile with CMP fields, and/or it may be a class that the Profile Developer chooses to expose through the Profile Local Interface of the Profile. A dependent value class can be the value of a CMP field.
    - The get accessor method for a CMP field that corresponds to a dependent value class returns a copy of the dependent value class instance.
    - The assignment of a dependent value class value to a CMP field using the set accessor method causes the value to be copied to the target CMP field.

        - A dependent value class must be Java serializable.
    - The CMP field accessor methods are mandatory transactional methods (see Section 9.6.1). They throw a `javax.slee.TransactionRequiredLocalException` if they are invoked without a valid transaction context.
    - Neither get nor set accessor methods are permitted to include a `throws` clause.
    - These methods may throw a `javax.slee.SLEEException` if the operation cannot be completed due to a system-level failure.
- The Profile CMP interface may have super-interfaces, however all super-interfaces are also subject to the rules for definition of Profile CMP interfaces

In the SLEE 1.0 specification, an SBB object was not permitted to invoke the set accessor methods on the Profile CMP interface. In SLEE 1.1 this is no longer necessarily the case. If a SLEE Component has a read-write view of a Profile (see Section 10.5.1.1) then the SLEE Component may invoke the set accessor methods on the Profile CMP interface to update the Profile. It is however recommended that SLEE Components use the Profile Local interface to perform this function in preference to the Profile CMP interface. If a SLEE Component has a read-only view of a Profile, then a SLEE Component that invokes a set accessor method on the Profile CMP interface should expect to receive a `java.lang.UnsupportedOperationException`, as was the case in SLEE 1.0[17].

10.6.1 Profile references in Profile CMP interface

A Profile may reference other Profiles (or itself) if it has attributes whose Java type is `ProfileID`, or Java types that contain `ProfileID` objects, such as classes that directly or indirectly contain `ProfileID` objects or `ProfileID` arrays. The SLEE does not check that the `ProfileID` objects in the CMP fields of a Profile refer to Profiles that exist.

Note: Future versions of the SLEE specification may include deployment descriptors that specify additional referential integrity constraints or checks for Profile references stored in CMP fields.

10.7 Profile Local interface

A Profile Local interface is the interface that SLEE Components use to invoke methods on a Profile. SLEE Components can invoke a target Profile in a synchronous manner through the Profile local interface of the target Profile. This interface is known as the Profile local interface because the object that represents the caller and the Profile object that represents the callee must be collocated in the same JVM. In order to be invoked synchronously, a Profile must declare a Profile local interface. The Profile local interface declares the methods of the Profile that may be invoked synchronously. A Profile local object is an instance of a SLEE implemented class that implements a Profile local interface.

10.7.1 How to obtain a Profile local object

A SLEE Component can obtain a Profile local object that represents a Profile via the following mechanisms:

- Receive the Profile local object as the result of invoking various methods on a `ProfileTable` object.
- An SBB component may retrieve a Profile local object from a SBB CMP field (see section 6.5).
- Receive the Profile local object as a result of calling various methods on a Profile event.
- Receive the Profile local object as a result of invoking a business method on a Profile local object. The callee may receive the Profile local object through one of its input arguments. The caller may receive the Profile local object through the return argument.

[17] The implementation of the Profile CMP interface could, for example, catch the `ReadOnlyProfileException` thrown by the Profile object and throw an `UnsupportedOperationException` in its place to the invoking SLEE Component.

10.7.2 What can be done with a Profile local object

A SLEE Component holding a reference to a Profile local object may do any of the following:

- Test if the Profile local object represents the same Profile as another Profile local object.
- Remove the Profile represented by the Profile local object.
- Invoke a Profile Specification Developer defined method of the Profile local object.
- Pass the Profile local object in an input or return argument of a local method call.
- Retrieve the name of the Profile.
- Retrieve the Profile Table for the Profile.

10.7.3 A typical scenario

In a typical scenario, an SBB object may consult provisioned data to perform a simple lookup, such as a number translation. In this scenario the "key" of the Profile is a non-translated number, and the SBB requests a Profile Table to return a matching result using a static query method defined on a Profile Table interface. The query method returns a Profile local object that represents a Profile matching the query. The calling SBB object then invokes a method on the query result to retrieve the translated number.

10.7.4 Profile Local interface specification

Each method invoked on the Profile Local interface causes a method with the same signature on the Profile abstract class to be invoked[18]. The transaction context of the caller of each method on the Profile Local interface is propagated to the corresponding method on the Profile abstract class.

The methods declared in a Profile Local interface are divided into the two following categories:

- CMP field get and/or set accessor methods.
  These methods are the same as those declared in the Profile CMP interface i.e. they have the same method signature. The SLEE provides the implementation of these methods in the Profile concrete class (see section 10.12).
- Business methods.
  These are methods declared in the Profile Local interface but not declared in the Profile CMP interface. The Profile abstract class must implement these business methods. The Profile Specification must include a Profile abstract class if business methods are included in the Profile Local interface. These business methods may invoke the CMP accessor methods.

The following are the requirements for the Profile Local interface:

- The Profile Local interface must be defined in a named package, i.e. the class must have a `package` declaration.
- The Profile Local interface must be declared as `public`.
- The Profile Local interface must extend `javax.slee.profile.ProfileLocalObject`.
- All methods have pass by reference semantics.
- All methods are mandatory transacted methods. If they are invoked without a valid transaction context then the `javax.slee.TransactionRequiredLocalException` is thrown.
- The Profile Local interface may extend the Profile CMP interface.
  If the Profile Local interface extends the Profile CMP interface, SLEE Components will be able to invoke the get and set accessor methods defined in the Profile CMP interface to read and modify all the attributes defined by the Profile CMP interface.

[18] Profile lifecycle methods may additionally be invoked so that a Profile object in the ready state services the invocation. For more information on the Profile object lifecycle please refer to section 10.13.2.

- Any Profile CMP field get and/or set accessor methods must have the same method signature as those declared in the Profile CMP interface. A subset of the Profile CMP interface accessor methods may be included in the Profile Local interface.
- The following rules apply to business methods:
  - Business methods may have any name, but are subject to the restrictions specified in Section 10.18.
  - The method arguments and return type must conform to the requirements specified in Section 10.17.
  - Business methods may not have the same name and arguments as a Profile CMP field get or set accessor method.
  - Methods may define a `throws` clause that throws any checked exception other than `java.rmi.RemoteException`. See section 10.7.5 for `RuntimeException` handling of Profile Local interfaces.
- All methods on the Profile Local interface throw the following exceptions:
  - `javax.slee.TransactionRequiredLocalException`
    This exception is thrown if the method is invoked without a valid transaction context.
  - `javax.slee.SLEEException`
    This exception is thrown if the operation cannot be performed due to a system level failure.
  - `javax.slee.TransactionRolledBackLocalException`
    This exception is thrown if the corresponding method on the Profile object throws a `RuntimeException`. For more information see section 10.7.5. This exception is also thrown if a method has been invoked on an invalid `ProfileLocalObject`, such as a `ProfileLocalObject` representing a Profile that no longer exists. The Profile Local objects returned by the various Profile events are an exception to this rule, as they reflect a snapshot in time of a Profile.
- The Profile Local interface may have super-interfaces, however all super-interfaces are also subject to the rules for definition of Profile Local interfaces.

The Profile Local interface is implemented by the SLEE. Methods which return a Profile Local interface return instances of a SLEE implemented class. If any method defined on the Profile Local interface is invoked, the method with the same signature is invoked on a Profile object (see section 10.13).

The Java class generated by the SLEE which implements the Profile Local interface is not guaranteed to be the same class as the Profile abstract class or the Profile concrete class. This means that the SBB Developer or Resource Adaptor Developer should not expect to be able to cast a Profile Local interface to a Profile abstract class or a Profile concrete class.

10.7.5 `RuntimeException` handling for the Profile Local interface

For each business method on the Profile Local interface there is a corresponding method in the Profile abstract class. If any method invoked on the Profile object in response to an invocation on the Profile Local interface returns by throwing an unchecked exception, the SLEE performs the following sequence of operations:

- The SLEE must log this occurrence.
- The SLEE marks the transaction of the invocation for rollback.
- The SLEE discards the Profile object.
- The SLEE throws a `javax.slee.TransactionRolledbackLocalException` to the caller of the Profile Local interface method. The runtime exception thrown by the Profile object

will be the cause of the `TransactionRolledbackLocalException` i.e. it is accessible via `java.lang.Throwable.getCause()`.

10.7.6 `ProfileLocalObject` interface

The `javax.slee.profile.ProfileLocalObject` interface declares the functionality that is common to all Profile Local interfaces.

The `ProfileLocalObject` interface is as follows:

```
package javax.slee.profile;

import javax.slee.NoSuchObjectLocalException;
import javax.slee.SLEEException;
import javax.slee.TransactionRequiredLocalException;

public interface ProfileLocalObject{
        public String getProfileName()
                throws SLEEException;

        public ProfileTable getProfileTable()
                throws SLEEException;

        public String getProfileTableName()
                throws SLEEException;

        public boolean isIdentical(ProfileLocalObject other)
                throws SLEEException;

        public void remove()
                throws  TransactionRequiredLocalException,
                        TransactionRolledbackLocalException,
                        SLEEException;
}
```

The following methods are defined:

- The `getProfileName` method.
  This method returns the name of the Profile represented by this `ProfileLocalObject`.
  This method is a non-transactional method.
  This method throws a `SLEEException` if the SLEE is unable to perform the operation due to a system-level failure.
- The `getProfileTable` method.
  This method returns the Profile Table of the Profile represented by this `ProfileLocalObject`. The return value is able to be type cast to the specific Profile Table interface defined by the Profile Specification.
  This method is a non-transactional method.
  This method throws a `SLEEException` if the SLEE is unable to perform the operation due to a system-level failure.
- The `getProfileTableName` method.
  This method returns the name of the Profile Table of the Profile represented by this `ProfileLocalObject`.
  This method is a non-transactional method.
  This method throws a `SLEEException` if the SLEE is unable to perform the operation due to a system-level failure.
- The `isIdentical` method.
  This methods returns true if the Profile represented by the argument is the same Profile as the Profile represented by this object.
  This method is a non-transactional method.

This method throws a `SLEEException` if the SLEE is unable to perform the operation due to a system-level failure.

- The `remove` method.
  This method attempts to remove the Profile represented by this `ProfileLocalObject`.
  This method is a mandatory transactional method. The profile is removed in the context of the calling transaction.
  This method throws the following exceptions:
  - `javax.slee.TransactionRequiredLocalException`.
    This exception is thrown if there is no transaction associated with the calling thread.
  - `javax.slee.TransactionRolledbackLocalException`.
    This exception is thrown when the Profile object returns by throwing an runtime exception. The runtime exception is the cause of this exception, i.e. it is accessible via `java.lang.Throwable.getCause()`. For example in the case where a Profile no longer exists, the `javax.slee.NoSuchObjectLocalException` is the cause of this exception. Another example is if the Profile Specification of the Profile Table was defined with a read-only SLEE Component view, and the `remove` method is invoked then the cause will be a `javax.slee.profile.ReadOnlyProfileException`.
  - `javax.slee.SLEEException`.
    This exception is thrown if the SLEE is unable to perform the operation due to a system-level failure.

10.8 Profile Table interface

The Profile Table interface defines methods to create, find, remove, and execute static queries on Profiles within a Profile Table. The `create`, `find`, and `remove` methods are defined by the `javax.slee.profile.ProfileTable` interface. Static query methods are defined on a Profile Specification Developer defined interface which extends the `ProfileTable` interface.

The Profile Specification Developer defined interface is the Profile Table interface for a Profile Specification which defines static queries that are able to be executed from the SLEE Component object view. The `javax.slee.profile.ProfileTable` interface is the Profile Table interface for a Profile Specification which does not provide the SLEE Component object view with any static queries.

The Profile Specification Developer defined Profile Table interface has the following requirements:

- The Profile Table interface must be defined in a named package, i.e. the class must have a `package` declaration.
- The Profile Table interface must be declared as `public`.
- The Profile Table interface must extend the `javax.slee.profile.ProfileTable` interface.
- For each static query defined in the Profile Specification there is a corresponding query method. The definition of a static query method is provided in Section 10.8.2

10.8.1 `ProfileTable` interface

The `javax.slee.profile.ProfileTable` interface declares the functionality that is common to all Profile Tables.

The `ProfileTable` interface is as follows:

```
package javax.slee.profile;

import java.util.Collection;
import javax.slee.SLEEException;
import javax.slee.TransactionRequiredLocalException;
import javax.slee.CreateException;
```

```
public interface ProfileTable {
        public ProfileLocalObject create(String profileName)
                throws  NullPointerException,
                        IllegalArgumentException,
                        TransactionRequiredLocalException,
                        ReadOnlyProfileException,
                        ProfileAlreadyExistsException,
                        CreateException,
                        SLEEException;

        public ProfileLocalObject find(String profileName)
                throws  NullPointerException,
                        TransactionRequiredLocalException,
                        SLEEException;

        public Collection findAll()
                throws TransactionRequiredLocalException,
                        SLEEException;

        public ProfileLocalObject findProfileByAttribute(
                                String attributeName, Object attributeValue)
                throws  NullPointerException,
                        IllegalArgumentException,
                        TransactionRequiredLocalException,
                        SLEEException;

        public Collection findProfilesByAttribute(
                                String attributeName, Object attributeValue)
                throws  NullPointerException,
                        IllegalArgumentException,
                        TransactionRequiredLocalException,
                        SLEEException;

        public boolean remove(String profileName)
                throws  NullPointerException,
                        ReadOnlyProfileException,
                        TransactionRequiredLocalException,
                        SLEEException;
}
```

- All methods on this interface are mandatory transactional methods.
- The `create` method creates a new Profile in the Profile Table with a Profile name specified by the `profileName` argument. This method returns a `ProfileLocalObject` which is able to be typecast to the Profile Local interface of the Profile Specification. The `ProfileLocalObject` returned by this method allows an SBB Developer or Resource Adaptor Developer to interact with the Profile within the same transaction context as it was created. If the transaction within which this method is invoked commits then the creation of the Profile is also committed. If the transaction within which this method is invoked rolls back then the creation of the Profile is also rolled back.

  This method throws the following exceptions:

  - `javax.slee.NullPointerException`.
    This exception is thrown if the `profileName` argument is `null`.
  - `java.lang.IllegalArgumentException`.
    This exception is thrown if the `profileName` argument does not match the rules laid out in section 10.2.4.
  - `javax.slee.profile.ReadOnlyProfileException`.
    This exception is thrown if the `profile-read-only` attribute of the Profile Specification is set to "True" (see section 10.5.1.1).

  - `javax.slee.profile.ProfileAlreadyExistsException`.
    This exception is thrown if a Profile with the specified `profileName` already exists in the Profile Table.
  - `javax.slee.CreateException`.
    This exception is propagated from the `profilePostCreate` method invoked on the `javax.slee.profile.Profile` interface as the Profile is created.

- The `find` method attempts to locate the Profile within the Profile Table identified by the `profileName` argument. If the Profile is found this method returns a `ProfileLocalObject` which is able to be typecast to the Profile Local interface of the Profile Specification. The `ProfileLocalObject` returned by this method allows the SBB Developer or Resource Adaptor Developer to interact with the Profile within the same transaction context as it was found. If the Profile is not found this method returns `null`. This method throws the following exception:
  - `javax.slee.NullPointerException`.
    This exception is thrown if the `profileName` argument is `null`.
- The `findAll` method returns all Profiles other than the default Profile within the Profile Table. This method returns a `java.util.Collection` object containing a `ProfileLocalObject` for each Profile found. Each `ProfileLocalObject` in the collection may be typecast to the Profile Local iInterface defined by the Profile Specification of the Profile Table. The collection returned by this method is immutable, i.e. elements may not be added to or removed from the collection. Any attempt to modify it, either directly or indirectly, will result in a `java.lang.UnsupportedOperationException` being thrown. If no Profiles exist in the Profile Table an empty collection is returned.
- The `findProfileByAttribute` method attempts to locate a Profile within the Profile Table where the Profile attribute identified by the `attributeName` argument is set to the value specified by the `attributeValue` argument. It is recommended that this method is only used for attributes with appropriate indexing (see section 10.22). This method can only be used on attributes whose Java type is a primitive type, an Object wrapper of primitive type (e.g. `java.lang.Integer`), `java.lang.String`, or `javax.slee.Address`. Equality for attributes of primitive types is specified by direct value comparison. Equality for object types is evaluated by `java.lang.Object.equals(Object)`. If more than one matching Profile is found the SLEE may arbitrarily return any one of the matching Profiles. The Profile Table's default Profile is not considered when determining matching Profiles. If no matching Profiles are found this method returns `null`. The `ProfileLocalObject` returned by this method may be safely typecast to the Profile Local interface defined by the Profile Specification of this Profile Table.
  This method may only be invoked on a Profile Table that was created from a SLEE 1.1 Profile Specification.
  This method throws the following exceptions:
  - `java.lang.NullPointerException`.
    This exception is thrown if any of the arguments to the method are `null`.
  - `java.lang.IllegalArgumentException`.
    This exception is thrown in the following scenarios: if an attribute with the specified name is not defined in the Profile Specification of the specified Profile Table, if the type of the `attributeValue` argument is different to the type of the Profile attribute, if the type of the attribute is not one of the allowed types.
  - `javax.slee.SLEEException`.
    This exception is thrown in the following cases: if an operation cannot be performed due to a system-level failure, or if the Profile Table was created from a SLEE 1.0 Profile Specification.

- The `findProfilesByAttribute` method attempts to locate all Profiles within the Profile Table where the Profile attribute identified by the attributeName argument is set to the value specified by the `attributeValue` argument. It is recommended that this method is only used for attributes with appropriate indexing (see section 10.22). This method can only be used on attributes whose Java type is a primitive type, an Object wrapper of primitive type (e.g. `java.lang.Integer`), `java.lang.String`, or `javax.slee.Address`. Equality for attributes of primitive types is specified by direct value comparison. Equality for object types is evaluated by `java.lang.Object.equals(Object)`. The Profile Table's default Profile is not considered when determining matching Profiles. This method returns a `Collection` object containing a `ProfileLocalObject` for each profile found. Each `ProfileLocalObject` in the collection may be typecast to the Profile Local interface defined by the Profile Specification of the Profile Table. The collection returned by this method is immutable, i.e. elements may not be added to or removed from the collection. Any attempt to modify it, either directly or indirectly, will result in a `java.lang.UnsupportedOperationException` being thrown. If no matching Profiles are found in the Profile Table an empty collection is returned.
  This method may only be invoked on a Profile Table that was created from a SLEE 1.1 Profile Specification.
  This method throws the following exceptions:
  - `java.lang.NullPointerException`.
    This exception is thrown if any of the arguments to the method are null.
  - `java.lang.IllegalArgumentException`.
    This exception is thrown in the following scenarios: if an attribute with the specified name is not defined in the Profile Specification of the specified Profile Table, if the type of the `attributeValue` argument is different to the type of the Profile attribute, if the type of the attribute is not one of the allowed types.
  - `javax.slee.SLEEException`.
    This exception is thrown in the following cases: if an operation cannot be performed due to a system-level failure, or if the Profile Table was created from a SLEE 1.0 Profile Specification.
- The `remove` method removes the Profile identified by the `profileName` argument from the Profile Table. This method returns `false` if no Profile existed in the Profile Table with the specified `profileName`. This method returns true if a Profile with the specified profileName existed in the Profile Table and was removed. If the transaction within which this method is invoked commits then the removal of the Profile is also committed. If the transaction within which this method is invoked rolls back then the removal of the Profile is also rolled back.
  This method throws the following exceptions:
  - `javax.slee.NullPointerException`.
    This exception is thrown if the `profileName` argument is `null`.
  - `javax.slee.profile.ReadOnlyProfileException`.
    This exception is thrown if the `profile-read-only` attribute of the Profile Specification is set to “True” (see section 10.5.1.1).
- All methods may throw the following exceptions:
  - `javax.slee.TransactionRequiredLocalException`.
    This exception is thrown if a method is invoked without a valid transaction context.
  - `javax.slee.SLEEException`.
    This exception is thrown if an operation cannot be performed due to a system-level failure.

10.8.2 Static query methods

The Profile Specification Developer declares a static query method for each static query in the Profile Specification which the Profile Specification Developer wants to have present in the SLEE Component object view. For more information on profile queries please refer to Section 10.20.

The method signature of this method is as follows:

```
public java.util.Collection query<<name>>(<<arguments>>)
       throws  TransactionRequiredLocalException,
               SLEEException;
```

- The method name must include the name of the static query that is specified in the query deployment descriptor element. The first letter of the name of the static query must be upper-cased and prefixed by "query".
- The number of method arguments must match the number of query-parameter elements defined in the query deployment descriptor element. The ordering and type of the method arguments must match the ordering and type of each query-parameter element.
- The return type is `java.util.Collection`. The returned `Collection` object contains a Profile Local object for each Profile that satisfies the query. Each Profile Local object in the collection may be typecast to the Profile Local interface of the Profile Specification. Each Profile Local object allows the Profile Specification Developer to interact with the corresponding Profile within the same transaction as the query was executed.
- Static query methods are mandatory transactional and may throw the following exceptions:
  - `javax.slee.TransactionRequiredLocalException`.
    This exception is thrown if this method is invoked without a valid transaction context.
  - `javax.slee.SLEEException`.
    This exception is thrown if the query operation cannot be performed due to a system-level failure.

The following is an example of a static query method that returns a Profile for the country where the country code is the longest matching prefix of a phone number.

The Profile CMP interface defined as:

```
package com.example;

public interface CountryPrefixProfileCMP{
…

    // an attribute representing a country code
    public String getCountryCode();
    public void setCountryCode(String code);

    // an attribute specifying the country name
    public String getCountryName();
    public void setCountryName(String name);

…
}
```

A static query is defined as:

```
<query name="matchingCountry">
       <query-parameter name="phNum" type="java.lang.String"/>
       <compare attribute="countryCode" op="longest-prefix-match " parameter="phNum"/>
</query>
```

The corresponding Profile Table interface is defined as:

```
package com.example;

import java.util.Collection;
import javax.slee.profile.ProfileTable;
```

```
public interface CountryPrefixProfileTable extends ProfileTable {
        public Collection queryMatchingCountry(String phoneNumber);
}
```

10.9 Profile Table Events and `ProfileTableActivity` objects

Each Profile Table has an associated Activity. The Java object that encapsulates this Activity is a `ProfileTableActivity` object. The SLEE fires the following types of events on a `ProfileTableActivity` object (if enabled by the `profile-events-enabled` attribute, see section 10.9.6).

- Profile Added Event.
  After a Profile is added to a Profile Table, the SLEE fires a Profile Added Event on the Profile Table's Activity.
- Profile Updated Event.
  After a Profile has been updated, the SLEE fires a Profile Updated Event on the Profile Table's Activity.
- Profile Removed Event.
  After a Profile is removed from a Profile Table, the SLEE fires a Profile Removed Event on the Profile Table's Activity.

These events are also known as the Profile Table Events.

Collectively, these events enable an SBB entity to be notified of additions, updates, and deletions to Profile Tables so that it can act on these changes with a valid transaction context. The SLEE invokes the event handler methods for these Profile Table Events within a transaction and these event handler methods can act on the Profile Table additions, updates, and deletions with full access to the runtime environment.

SBBs register interest in these events like any other events using `event` elements in their deployment descriptors and by attaching to the appropriate Activity Contexts.

10.9.1 `ProfileTableActivity` objects

The SLEE fires Profile Table Events on `ProfileTableActivity` objects.

- The SLEE is the creator and owner of all `ProfileTableActivity` objects.
- At any point in time, a Profile Table has at most one `ProfileTableActivity` object. If a `ProfileTableActivity` object does not exist when required, for example to fire a Profile Table Event, it is created by the SLEE.
- When a Profile Table is removed from the SLEE by an Administrator using a `ProfileProvisioningMBean` object, the SLEE ends the corresponding Profile Table Activity and fires an Activity End Event on its `ProfileTableActivity` object. Additionally, the SLEE ends all Profile Table Activities, and fires Activity End Events on the corresponding `ProfileTableActivity` objects, when the SLEE transitions from the Running state to the Stopping state. This allows SBB entities attached to Profile Table Activities to clean up in the usual way.

The `ProfileTableActivity` interface is as follows:

```
package javax.slee.profile;

public interface ProfileTableActivity {
        public String getProfileTableName();
}
```

10.9.2 Profile Added Events

The `event-definition` element for Profile Added Events is as follows:

```
<event-definition>
```

```
        …
        <event-type-name> javax.slee.profile.ProfileAddedEvent </event-type-name>
        <event-type-vendor> javax.slee </event-type-vendor>
        <event-type-version> 1.0 </event-type-version>
        <event-class-name> javax.slee.profile.ProfileAddedEvent </event-class-name>
</event-definition>
```

The `ProfileAddedEvent` interface is as follows:

```
package javax.slee.profile;

import javax.slee.Address;

public interface ProfileAddedEvent {
        public Address getProfileAddress();
        public ProfileID getProfile();
        public ProfileLocalObject getAddedProfileLocal();

        // deprecated
        public Object getAddedProfile();
}
```

- The `getProfileAddress` method returns an `Address` object identifying the Profile that has been added. When an event is fired, the event producer may optionally provide to the event router a default address that is used to route the event (see Section 8.2.1). In the case of a Profile Added Event, the SLEE is the event producer and the default address provided by the SLEE for the Profile Added Event is the address returned by this method.
- The `getProfile` method returns the Profile identifier of the Profile added to the Profile Table. The Profile identifier may or may not identify an existing Profile in the Profile Table. For example, the Profile identifier does not identify an existing Profile in the Profile Table if the Profile has been deleted after it was added but before the SLEE delivers the event.
- The `getAddedProfileLocal` method returns a Profile Local object that can be used to access the content of the Profile when the Profile was added.
    - The Java type of the Profile Local object is the Profile Local interface defined by Profile Specification of the Profile Table identified by the Profile identifier returned by the `getProfile` method. Therefore, Java typecast can be used to narrow the returned object to the Profile Local interface or a base interface of the Profile Local interface.
    - The values in the attributes abstracted by this Profile Local object are the values present in the Profile when the Profile was added to the Profile Table. These values may be different from the current attribute values of the same Profile. The values may be different if the Profile identified by the Profile identifier has been modified after the Profile was added.
    - The Profile Local object returned by an invocation of this method is only valid within the same transaction that the invocation occurred in. (The same event may be re-fired and processed in a new transaction, if this method is invoked to obtain another Profile Local object, this new Profile Local object will only be valid within the new transaction.)
    - This method may return `null` if the relevant Profile Specification implementation classes for the Profile are not available in the classloader of an SBB which receives this event. This may occur, for example, if an SBB entity attaches to the Profile Table Activity of a Profile Table for which the SBB of the SBB entity has not declared a `profile-spec-ref` entry in its deployment descriptor for the Profile Specification of the Profile Table.
    - As the Profile Local object represents a snapshot in time of the Profile state, rather than a Profile that exists in the Profile Table, the Profile Local object exhibits a read-only view of the Profile with the same behavior as defined in Section 10.5.1.1.

- The `getAddedProfile` method is deprecated in SLEE 1.1. It returns a read-only Profile CMP object that provides the content of the Profile when the Profile was added. It is recommended that the `getAddedProfileLocal` method be used instead of this method.
    - The Java type of the Profile CMP object is the Profile CMP interface defined by Profile Specification of the Profile Table identified by the Profile identifier returned by the `getProfile` method. Therefore, Java typecast can be used to narrow the returned object to the Profile CMP interface or a base interface of the Profile CMP interface.
    - The values in the attributes of this Profile CMP object are the values present in the Profile when the Profile was added to the Profile Table. These values may be different from the current attribute values of the same Profile. The values may be different if the Profile identified by the Profile identifier has been modified after the Profile was added.
    - The Profile CMP object returned by an invocation of this method is only valid within the same transaction that the invocation occurred in. (The same event may be re-fired and processed in a new transaction, if this method is invoked to obtain another Profile CMP object, this new Profile CMP object will only be valid within the new transaction.)
    - This method may return `null` if the relevant Profile Specification implementation classes for the Profile are not available in the classloader of an SBB which receives this event. This may occur, for example, if an SBB entity attaches to the Profile Table Activity of a Profile Table for which the SBB of the SBB entity has not declared a `profile-spec-ref` entry in its deployment descriptor for the Profile Specification of the Profile Table.

10.9.3 Profile Updated Events

The `event-definition` element for Profile Updated Events is as follows:

```
<event-definition>
        …
        <event-type-name> javax.slee.profile.ProfileUpdatedEvent </event-type-name>
        <event-type-vendor> javax.slee </event-type-vendor>
        <event-type-version> 1.0 </event-type-version>
        <event-class-name> javax.slee.profile.ProfileUpdatedEvent </event-class-name>
</event-definition>
```

The `ProfileUpdatedEvent` interface is as follows:

```
package javax.slee.profile;

import javax.slee.Address;

public interface ProfileUpdatedEvent {
        public Address getProfileAddress();
        public ProfileID getProfile();
        public ProfileLocalObject getBeforeUpdateProfileLocal();
        public ProfileLocalObject getAfterUpdateProfileLocal();

        // deprecated
        public Object getBeforeUpdateProfile();
        public Object getAfterUpdateProfile();
}
```

- The `getProfileAddress` method returns an `Address` object that identifies the Profile that has been updated. When an event is fired, the event producer may optionally provide to the event router a default address (see Section 8.2.1). In the case of a Profile Updated Event, the default address is the address returned by this method.
- The `getProfile` method returns the Profile identifier of the Profile updated. The Profile identifier may or may not identify an existing Profile in the Profile Table. For example, the Profile identifier may not identify an existing Profile in the Profile Table if the Profile has been removed after the Profile was updated but before the SLEE delivers the event.

- The `getBeforeUpdateProfileLocal` method and the `getAfterUpdateProfileLocal` method return Profile Local objects can be used to access the content of the Profile before and after the Profile was updated, respectively.
    - The Java type of the Profile Local object is the Profile Local interface defined by Profile Specification of the Profile Table identified by the Profile identifier returned by the `getProfile` method. Therefore, Java typecast can be used to narrow the returned object to the Profile Local interface or a base interface of the Profile Local interface.
    - The values in the attributes abstracted by the Profile Local object returned by `getBeforeUpdateProfileLocal` and `getAfterUpdateProfileLocal` are the values in the Profile immediately before and after the Profile was updated, respectively. These values may be different from the current attribute values for the same Profile. The values may be different if the Profile identified by the Profile identifier has been modified again since the update that was reported by the event.
    - The Profile Local object returned by an invocation of one of these methods is only valid within the same transaction that the invocation occurred in. (The same event may be re-fired and processed in a new transaction, if one of these method is invoked to obtain another Profile Local object, this new Profile Local object will only be valid within the new transaction.)
    - These methods may return `null` if the relevant Profile Specification implementation classes for the Profile are not available in the classloader of an SBB which receives this event. This may occur, for example, if an SBB entity attaches to the Profile Table Activity of a Profile Table for which the SBB of the SBB entity has not declared a `profile-spec-ref` entry in its deployment descriptor for the Profile Specification of the Profile Table.
    - As the Profile Local object represents a snapshot in time of the Profile state, rather than a Profile that exists in the Profile Table, the Profile Local object exhibits a read-only view of the Profile with the same behavior as defined in Section 10.5.1.1.
- The `getBeforeUpdateProfile` method and the `getAfterUpdateProfile` method are deprecated in SLEE 1.1. They return read-only Profile CMP objects that provide the content of the Profile before and after the Profile was updated, respectively. It is recommended that the `getBeforeUpdateProfileLocal` and `getAfterUpdateProfileLocal` methods be used instead of these methods.
    - The Java type of the Profile CMP object is the Profile CMP interface defined by Profile Specification of the Profile Table identified by the Profile identifier returned by the `getProfile` method. Therefore, Java typecast can be used to narrow the returned object to the Profile CMP interface or a base interface of the Profile CMP interface.
    - The values in the attributes of the Profile CMP object returned by `getBeforeUpdateProfile` and `getAfterUpdateProfile` are the values in the Profile immediately before and after the Profile was updated, respectively. These values may be different from the current attribute values of the same Profile. The values may be different if the Profile identified by the Profile identifier has been modified again since the update that was reported by the event.
    - The Profile CMP object returned by an invocation of one of these methods is only valid within the same transaction that the invocation occurred in. (The same event may be re-fired and processed in a new transaction, if one of these method is invoked to obtain another Profile CMP object, this new Profile CMP object will only be valid within the new transaction.)
    - These methods may return `null` if the relevant Profile Specification implementation classes for the Profile are not available in the classloader of an SBB which receives this event. This may occur, for example, if an SBB entity attaches to the Profile Table

Activity of a Profile Table for which the SBB of the SBB entity has not declared a `profile-spec-ref` entry in its deployment descriptor for the Profile Specification of the Profile Table.

10.9.4 Profile Removed Events

The `event-definition` element for Profile Removed Events is as follows:

```
<event-definition>
        …
        <event-type-name> javax.slee.profile.ProfileRemovedEvent </event-type-name>
        <event-type-vendor> javax.slee </event-type-vendor>
        <event-type-version> 1.0 </event-type-version>
        <event-class-name> javax.slee.profile.ProfileRemovedEvent </event-class-name>
</event-definition>
```

The `ProfileRemovedEvent` interface is as follows:

```
package javax.slee.profile;

import javax.slee.Address;

public interface ProfileRemovedEvent {
        public Address getProfileAddress();
        public ProfileID getProfile();
        public ProfileLocalObject getRemovedProfileLocal();

        // deprecated
        public Object getRemovedProfile();
}
```

- The `getProfileAddress` method returns an `Address` object identifying the Profile that has been removed. When an event is fired, the event producer may optionally provide to the event router a default address that is used to route the event (see Section 8.2.1). In the case of a Profile Removed Event, the SLEE is the event producer and the default address provided by the SLEE for the Profile Removed Event is the address returned by this method.
- The `getProfile` method returns the Profile identifier of the Profile removed from the Profile Table. The Profile identifier may or may not identify an existing Profile in the Profile Table. For example, the Profile identifier may identify a new Profile in the Profile Table if a new Profile has been added with the same Profile identifier after the previous Profile with same Profile identifier was removed but before the SLEE delivers the event.
- The `getRemovedProfileLocal` method returns a Profile Local object that can be used to access the content of the Profile when the Profile was removed.
    - The Java type of the Profile Local object is the Profile Local interface defined by Profile Specification of the Profile Table identified by the Profile identifier returned by the `getProfile` method. Therefore, Java typecast can be used to narrow the returned object to the Profile Local interface or a base interface of the Profile Local interface.
    - The values in the attributes abstracted by the Profile Local object are the values present in the Profile immediately before the Profile was removed from the Profile Table. These values may be different from the current attribute values for a Profile with the same Profile identifier. The values may be different if the Profile identified by the Profile identifier has been added after the previous Profile identified by the Profile identifier was removed.
    - The Profile Local object returned by an invocation of this method is only valid within the same transaction that the invocation occurred in. (The same event may be re-fired and processed in a new transaction, if this method is invoked to obtain another Profile Local object, this new Profile Local object will only be valid within the new transaction.)

  - This method may return `null` if the relevant Profile Specification implementation classes for the Profile are not available in the classloader of an SBB which receives this event. This may occur, for example, if an SBB entity attaches to the Profile Table Activity of a Profile Table for which the SBB of the SBB entity has not declared a `profile-spec-ref` entry in its deployment descriptor for the Profile Specification of the Profile Table.
  - As the Profile Local object represents a snapshot in time of the Profile state, rather than a Profile that exists in the Profile Table, the Profile Local object exhibits a read-only view of the Profile with the same behavior as defined in Section 10.5.1.1.

- The `getRemovedProfile` method is deprecated in SLEE 1.1. It returns a read-only Profile CMP object that provides the content of the Profile when the Profile was removed. It is recommended that the `getRemovedProfileLocal` method be used instead of this method.
  - The Java type of the Profile CMP object is the Profile CMP interface defined by Profile Specification of the Profile Table identified by the Profile identifier returned by the `getProfile` method. Therefore, Java typecast can be used to narrow the returned object to the Profile CMP interface or a base interface of the Profile CMP interface.
  - The values in the attributes of the Profile CMP object are the values present in the Profile immediately before the Profile was removed from the Profile Table. These values may be different from the current attribute values for a Profile with the same Profile identifier. The values may be different if the Profile identified by the Profile identifier has been added after the previous Profile identified by the Profile identifier was removed.
  - The Profile CMP object returned by an invocation of this method is only valid within the same transaction that the invocation occurred in. (The same event may be re-fired and processed in a new transaction, if this method is invoked to obtain another Profile CMP object, this new Profile CMP object will only be valid within the new transaction.)
  - This method may return `null` if the relevant Profile Specification implementation classes for the Profile are not available in the classloader of an SBB which receives this event. This may occur, for example, if an SBB entity attaches to the Profile Table Activity of a Profile Table for which the SBB of the SBB entity has not declared a `profile-spec-ref` entry in its deployment descriptor for the Profile Specification of the Profile Table.

10.9.5 `ProfileTableActivityContextInterfaceFactory` interface

Changed in 1.1: JNDI location constant added.

An SBB can obtain a generic Activity Context Interface object for a Profile Table Activity using a `ProfileTableActivityContextInterfaceFactory` object. The `ProfileTableActivityContextInterfaceFactory` object is bound to a SLEE specification defined location in the component environment of every SBB (see Section 13.9).

```
package javax.slee.profile;

import javax.slee.ActivityContextInterface;
import javax.slee.FactoryException;
import javax.slee.UnrecognizedActivityException;
import javax.slee.TransactionRequiredLocalException;

public interface ProfileTableActivityContextInterfaceFactory {
       // JNDI location constant
       public static final String JNDI_NAME =
         "java:comp/env/slee/facilities/profiletableactivitycontextinterfacefactory";

       public ActivityContextInterface
              getActivityContextInterface(ProfileTableActivity tableActivity)
                     throws  NullPointerException,
                             TransactionRequiredLocalException,
```

```
                              UnrecognizedActivityException,
                              FactoryException;
}
```

- The `JNDI_NAME` constant. *(Added in 1.1)*
  This constant specifies the JNDI location where a `ProfileTableActivityContextInterfaceFactory` object may be located by an SBB component in its component environment.
- See Section 7.6.1 for a general description on Activity Context Interface Factories, the transactional semantics, and the common exceptions thrown by a `getActivityContextInterface` method.

10.9.6 Profile Events Enabled Attribute

A `profile-events-enabled` boolean attribute defined in the `profile-spec` element of a Profile Specification's deployment descriptor determines whether a Profile event is generated when a Profile is created, deleted or modified in a Profile Table created from the Profile Specification. The default value of this attribute is "True". This default value retains backwards compatibility with SLEE 1.0 Profiles (where each modification causes an event to be fired).

If the `profile-events-enabled` attribute is set to "False" the SLEE will not generate Profile events for any Profile Tables created from this Profile Specification. This is an addition to the SLEE 1.0 specification. An application may not want Profile events because the application level logic is designed to detect changes through other means. For example, the application may detect changes via periodic query of tables for rows with update times later than the last query.

10.10 Profile Management interface

The Profile Management interface is the interface that defines the methods made available to management clients. When a Profile Specification is deployed the SLEE implements a Profile MBean that contains the methods defined in the Profile Management interface. A management client interacts with a Profile using a Profile MBean object (see Section 10.26).

The methods declared in a Profile Management interface are divided into the two following categories:

- CMP field get and set accessor methods.
  These methods are the same as those declared in the Profile CMP interface. The Profile Specification Developer does not provide the implementations of these methods in the Profile abstract class. The SLEE provides the implementations of these methods in a concrete subclass of the Profile abstract class.
- Management methods.
  These methods are declared in the Profile Management interface and not declared in the Profile CMP interface. The Profile abstract class must implement these business methods. The Profile Specification must include a Profile abstract class if business methods are included in the Profile Management interface. These business methods may for example update a set of Profile attributes in one go or return some result computed from the values of a set of attributes.

The following are the requirements for the Profile Specification Developer defined Profile Management interface:

- The Profile Management interface must be defined in a named package, i.e. the class must have a `package` declaration.
- The Profile Management interface must be declared as `public`.
- The Profile Management interface may extend the Profile CMP interface.
  If the Profile Management interface extends the Profile CMP interface, management clients will be able to invoke the get and set accessor methods defined in the Profile CMP interface to read and modify all the attributes defined by the Profile CMP interface.

- If the Profile Management interface does not extend the Profile CMP interface, the Profile Management interface may still declare CMP field get and set accessor methods. The method name, argument and return types of these methods must be the same as declared in the Profile CMP interface and must not have a `throws` clause. These methods are not considered to be management methods as they will be implemented by the SLEE.
- The Profile Management interface may declare management methods.
  The following rules apply to management methods:
  - Management methods may have any name, but are subject to the restrictions specified in Section 10.18.
  - The method arguments and return type must conform to the requirements specified in Section 10.17.
  - Management methods may not have the same name and arguments as a Profile CMP field get or set accessor method.
  - Methods may define an arbitrary `throws` clause.
- The Profile Management interface may have super-interfaces, however all super-interfaces are also subject to the rules for definition of Profile Management interfaces.

10.11 Profile abstract class

Note: The Profile abstract class replaces the Profile Management abstract class from the SLEE 1.0 Specification.

The Profile abstract class is optional in the Profile Specification, however a Profile abstract class must be defined if any of the following cases are true:

- Management methods are defined in the Profile Management interface
- Business methods are defined in the Profile Local interface.

The following are the requirements for the Profile abstract class:

- The Profile abstract class must be defined in a named package, i.e. the class must have a `package` declaration.
- The Profile abstract class must be declared as `public` and `abstract`.
- The Profile abstract class must have a public constructor that takes no arguments and has no `throws` clause.
- The Profile abstract class implements the following interfaces:
  - The Profile abstract class must be defined to implement the Profile CMP interface, however the methods declared in the Profile CMP interface must remain abstract in the Profile abstract class. The get and set accessor methods of the Profile CMP interface are implemented by the SLEE at deployment time when it generates the Profile concrete class (see Section 10.12)
  - The Profile abstract class must implement the `javax.slee.profile.Profile` interface and must provide an implementation of all methods defined in the `Profile` interface. For example, the lifecycle callback methods allow initial values for Profile attributes to be set when the default Profile is created, or to perform verification checks on Profile attribute values when changes are made to a Profile by the Administrator.
  - If the Profile Specification defines a distinct Profile Local interface (i.e. a Profile Local interface that is not the same as the Profile CMP interface) then the following rules apply:
    - Each business method in the Profile Local interface must have a matching method with the same signature in the Profile abstract class. The matching method in the Profile abstract class provides the implementation logic.

        - These method implementations must be `public`, and cannot be `static`, `abstract`, or `final`.
        - The Profile abstract class must not include the Profile Local interface in its list of implemented interfaces.
    - If the Profile Specification defines a distinct Profile Management interface (i.e. a Profile Management interface that is not the same as the Profile CMP interface), then the following rules apply:
        - The Profile abstract class must implement the Profile Management interface. All management methods defined in the Profile Management interface must be implemented in the Profile abstract class.
        - These method implementations must be `public`, and cannot be `static`, `abstract`, or `final`.
- If the Profile Specification Developer defines a Usage Parameters interface, then the Profile abstract class must declare at least one of the two get Usage Parameters methods. Each of these methods returns an object that implements the Usage Parameters interface of the Profile Specification. This allows the Profile to manipulate the usage parameters named by the Profile Specification (see Section 11.4.2).

The Profile abstract class may have super-classes, however all super-classes are also subject to the rules for definition of Profile abstract classes.

10.11.1 Non-reentrant and reentrant Profiles

A Profile Specification Developer can specify that Profiles of a Profile Specification are non-reentrant. If a Profile of a non-reentrant Profile Specification is executing in a given transaction context, and another method invocation with the same transaction context arrives for the same Profile, the SLEE will throw an exception to the second request. This rule allows the Profile Specification Developer to program the Profile abstract class as single-threaded non-reentrant code.

The functionality of a Profile may require loop backs in the same transaction context. An example of a loop back is when Profile A receives a method invocation, A calls Profile B, and B calls back A in the same transaction context. The method invoked by the loop back shares the current execution context (which includes the transaction) with the initial invocation of Profile A.

If the Profile abstract class is specified as non-reentrant in the deployment descriptor, the SLEE must reject an attempt to reenter a Profile using that Profile abstract class while the Profile is executing a method invocation. (This can happen, for example, if the Profile has invoked another Profile and the other Profile tries to make a loop back call.) If the attempt is made to reenter the same Profile, the SLEE must throw the `javax.slee.SLEEException` to the caller. The SLEE must allow the call if the Profile Specification's deployment descriptor specifies that the Profile abstract class is reentrant.

Reentrant Profile abstract classes must be programmed and used with caution. First, the Profile Specification Developer must code the Profile abstract class with the anticipation of a loop back call.

Profile Specifications that do not need callbacks should be marked as non-reentrant in the deployment descriptor, allowing the SLEE to detect and prevent illegal concurrent calls.

The following snippet of a Profiles deployment descriptor shows how to specify reentrant behaviour.

```
<profile-spec-jar>
...
            <profile-abstract-class reentrant="True">
...
            </profile-abstract-class>
...
</profile-spec-jar>
```

10.11.2 Non-persistent or transient state

A Profile abstract class may use instance variables to maintain transient state, or to store values that depend on the persistent state cached by the CMP fields (or vice-versa). The `profileLoad` method (see Section 10.13.1.8) can be used by the Profile abstract class to resynchronize the values stored in the instance variables that depend on the persistent state cached by the CMP fields. In general, any transient state that depends on the persistent state cached by the CMP fields should be recomputed during the `profileLoad` method. CMP fields whose value depends on transient state stored in instance variables should have their values set during the `profileStore` method (see Section 10.13.1.9). The instance variables of a Profile abstract class are not guaranteed to retain their values after `profileStore` has returned.

10.12 Profile concrete class

A Profile concrete class is implemented by the SLEE when a Profile Specification is deployed. The Profile concrete class extends the Profile abstract class and implements the Profile CMP methods.

The following rules apply to the Profile concrete class implemented by the SLEE:

- If a Profile abstract class is defined, then the SLEE implemented Profile concrete class extends the Profile abstract class and implements the Profile CMP methods.
- If a Profile abstract class is not defined, then the SLEE implemented Profile concrete class provides an implementation of the Profile CMP interface.

10.13 Profile objects

A Profile object is an instance of a Profile concrete class. The SLEE instantiates a Profile object to interact with the data maintained in a Profile on behalf of a management client or a SLEE Component. A Profile object is the runtime executable object that caches and manipulates the persistent data in a Profile. The management client or SLEE Component does not have direct access to the Profile object. A management client may interact with a Profile object via a Profile MBean object (see Section 10.26). A SLEE Component may interact with a Profile via its Profile Local interface (see Section 10.7).

10.13.1 `Profile` interface

Note: The `Profile` *interface replaces the* `ProfileManagement` *interfaces from the SLEE 1.0 Specification.*

The `javax.slee.profile.Profile` interface declares the callback methods that apply to Profile objects. This interface defines various life cycle callback methods. The life cycle of a Profile object is further discussed in Section 10.13.2.

The `Profile` interface is as follows:

```
package javax.slee.profile;

import javax.slee.CreateException;

public interface Profile {
        public void setProfileContext(ProfileContext context);
        public void unsetProfileContext();
        public void profileInitialize();
        public void profilePostCreate()
                throws CreateException;
        public void profileRemove();
        public void profileActivate();
        public void profilePassivate();
        public void profileLoad();
        public void profileStore();
        public void profileVerify()
                throws ProfileVerificationException;
}
```

10.13.1.1 `setProfileContext` method

The `Profile` interface defines the method signature of the `setProfileContext` method as follows:

```
public void setProfileContext(ProfileContext context);
```

The SLEE invokes this method after a new instance of the Profile concrete class is created. It uses this method to pass a `ProfileContext` object to the Profile object. If the Profile object needs to use the `ProfileContext` object during its lifetime, it must store the `ProfileContext` object in an instance variable.

The Profile object can take advantage of this method to allocate and initialize state held by the Profile object during its lifetime. Since a Profile has not been assigned to the Profile object when this method is invoked, such state and resources cannot be specific to a particular Profile because the Profile object might be reused during its lifetime to serve multiple Profiles.

- The `setProfileContext` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The Profile object must not attempt to access its persistent state using the Profile CMP interface methods during this method. Attempting to do so causes a `java.lang.IllegalStateException` to be thrown.
- This method is invoked with an unspecified transaction context (see Section 9.7.1 for details on how the SLEE executes methods with an unspecified transaction context).

10.13.1.2 `unsetProfileContext` method

The `Profile` interface defines the method signature of the `unsetProfileContext` method as follows:

```
public void unsetProfileContext();
```

The SLEE invokes this method before releasing its references to the Profile object[19]. During this method, a Profile is not assigned to the Profile object. The Profile object can take advantage of this method to free any state held by the Profile object. This state has typically been allocated by the `setProfileContext` method.

- The `unsetProfileContext` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The Profile object must not attempt to access its persistent state using the Profile CMP interface methods during this method. Attempting to do so causes a `java.lang.IllegalStateException` to be thrown.
- This method is invoked with an unspecified transaction context (see Section 9.7.1 for details on how the SLEE executes methods with an unspecified transaction context).

10.13.1.3 `profileInitialize` method

The `Profile` interface defines the method signature of the `profileInitialize` method as follows:

```
public void profileInitialize();
```

The SLEE invokes this method on a Profile object in order to populate the CMP state of the default Profile. The implementation of this method should initialize the default Profile by using the Profile CMP interface methods.

- The `profileInitialize` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.

[19] Once the SLEE releases its references, the Profile object is a candidate for Java object Garbage Collection.

- The SLEE guarantees that the values that will be initially returned by the Profile CMP interface get methods will be the Java language defaults (e.g. 0 for integer, `null` for object references).
- The SLEE invokes this method with a transaction context (see Section 9.6). The default Profile is created in the same transaction context as the `profileInitialize` method. After this method returns, the SLEE creates the default Profile.

10.13.1.4 `profilePostCreate` method

The `Profile` interface defines the method signature of the `profilePostCreate` method as follows:

```
public void profilePostCreate() throws javax.slee.CreateException;
```

The SLEE invokes this method on a Profile object *after* the SLEE creates a new Profile. The SLEE invokes this method after the persistent representation of the Profile has been created and the Profile object is assigned to the newly created Profile. This method gives the Profile object a chance to initialize additional transient state and acquire additional resources that it needs while it is in the Ready state.

- The `profilePostCreate` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- This method may throw a `javax.slee.CreateException` when there is an application level problem (rather than SLEE or system-level problem). The SLEE will propagate the `CreateException` to the caller that requested the creation of the Profile. In the case that the caller is a SLEE component (e.g. SBB or Resource Adaptor) and the caller is using the `create` method of the `ProfileTable` interface then the exception is propagated unchanged. In the case that the caller is a management client then a `javax.slee.CreateException` is thrown to the client – the client will receive "the same exception"[20].
- The SLEE invokes this method with the transaction context (see Section 9.6) used to create the Profile. In the case that the client creating the Profile is a SLEE component (e.g. an SBB or Resource Adaptor) then the `profilePostCreate` method executes in the caller's transaction context. In the case that the client is a management client then the `profilePostCreate` method executes in the transaction associated started for the purpose of creating the profile.
- The Profile object enters the Ready state after `profilePostCreate` returns normally. If `profilePostCreate` returns by throwing an exception, the Profile object does not enter the Ready state.

10.13.1.5 `profileActivate` method

The `Profile` interface defines the method signature of the `profileActivate` method as follows:

```
public void profileActivate();
```

The SLEE invokes this method on a Profile object when the SLEE wants to assign a specific Profile to a Profile object in the Pooled state. This method gives the Profile object a chance to initialize additional transient state that it needs while it is in the Ready state.

- The `profileActivate` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The Profile object must not attempt to access the Profile's persistent state using the Profile CMP interface methods during this method. Attempting to do so causes a `java.lang.IllegalStateException` to be thrown.
- This method executes with an unspecified transaction context.

[20] This means that the exception received by the management client has the same message and stack trace etc, but may be a different Java object.

10.13.1.6 `profilePassivate` method

The `Profile` interface defines the method signature of the `profilePassivate` method as follows:

```
public void profilePassivate();
```

The SLEE invokes this method on a Profile object when the SLEE decides to disassociate the Profile object from the Profile, and to return the Profile object to the Pooled state. This method gives the Profile object the chance to release any state or resources that should not be held while the Profile object is in the Pooled state. This state has typically had been allocated during the `profileActivate` method.

- The `profilePassivate` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The Profile object must not attempt to access the Profile's persistent state using the Profile CMP interface methods during this method. Attempting to do so causes a `java.lang.IllegalStateException` to be thrown.
- This method executes with an unspecified transaction context.

10.13.1.7 `profileRemove` method

The `Profile` interface defines the method signature of the `profileRemove` method as follows:

```
public void profileRemove();
```

The SLEE invokes the `profileRemove` method on a Profile object before the SLEE removes the Profile which the Profile object is assigned to. The SLEE removes a Profile when the Profile is removed explicitly by an invocation of the `remove` method on the `ProfileTable` interface, or by the `removeProfile` method on the `ProfileProvisioningMBean` interface. The Profile object is in the Ready state when `profileRemove` is invoked and it will enter the Pooled state after the method returns. The Profile Specification Developer can use this method to implement any actions that must be done before the Profile's persistent representation is removed.

- The `profileRemove` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The SLEE synchronizes the Profile object's state before it invokes the `profileRemove` method. This means that the Profile CMP state of the Profile object at the beginning of this method is the same as it would be at the beginning of a local interface method or management method.
- This method is invoked within a transaction context. In the case that the client removing the Profile is a SLEE Component then this method executes in the caller's transaction context. In the case that the client is a management client the this method executes in the transaction started for the purpose of removing the profile. After this method returns, and in the same transaction context, the SLEE executes the internal operations that delete the persistent representation of the Profile.
- Since the Profile object will enter the Pooled state, the state of the Profile object at the end of this method must be equivalent to the state of a passivated Profile object. This means that the Profile object must free any state and release any resource that it would normally release in the `profilePassivate` method.

10.13.1.8 `profileLoad` method

The `Profile` interface defines the method signature of the `profileLoad` method as follows:

```
public void profileLoad();
```

The SLEE calls this method to synchronize the state of a Profile object with its assigned Profile's persistent state. The Profile Specification Developer can assume that the Profile object's persistent state has been loaded just before this method is invoked. It is the responsibility of the Profile Specification Developer to

use this method to re-compute or initialize the values of any transient instance variables that depend on the Profile's persistent state. In general, any transient state that depends on the persistent state of a Profile should be recalculated in this method. The Profile Specification Developer can use this method, for instance, to perform some computation on the values returned by the CMP field accessor methods, such as converting text fields to more convenient objects or binary representations.

- The `profileLoad` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The SLEE invokes this method within a transaction context.

10.13.1.9 `profileStore` method

The `Profile` interface defines the method signature of the `profileStore` method as follows:

```
public void profileStore();
```

The SLEE calls this method to synchronize the state of the Profile's persistent state with the state of the Profile object. The Profile Specification Developer should use this method to update the Profile's persistent state using the CMP field accessor methods. For example, this method may perform conversion of object or binary data representations to text. The Profile Specification Developer can assume that after this method returns, the persistent state is synchronized.

- The `profileStore` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- The SLEE invokes this method within a transaction context.

10.13.1.10 `profileVerify` method

Changed in 1.1: This method is not called on the default Profile for 1.1 Profile Specifications.

The `Profile` interface defines the method signature of the `profileVerify` method as follows:

```
public void profileVerify()
        throws javax.slee.profile.ProfileVerificationException;
```

The SLEE invokes this method on a Profile object after the management client has invoked the `commitProfile` method on a Profile MBean (see Section 10.26.3.2). This method enables the Profile to reject changes made by the management client where those changes are considered invalid. The Profile rejects such changes by throwing a `ProfileVerificationException`.

- The `profileVerify` method must be declared as `public` and cannot be `static`, `abstract`, or `final`.
- This method is invoked on a Profile object in the Ready state.
- This method may throw the `javax.slee.profile.ProfileVerificationException` if it detects that the Profile CMP state is invalid. This exception is propagated to the management client that invoked the `commitProfile` method on the Profile MBean.
- The SLEE invokes this method in a transaction context. The `profileVerify` method must run after the `profileStore` method, and in the same transaction context as that begun by the `editProfile` invocation (on the `ProfileMBean` interface).

This method is not invoked for any transaction which is originated by the SLEE during event delivery or as a consequence of Profile Local method invocations by any SLEE component (e.g. SBB or Resource Adaptor).

In addition, this method is not invoked for any changes made by the Administrator to the default profile of a Profile Table as the default Profile may contain values that are not valid for a named Profile, e.g. an attribute of the default Profile might have a `null` value even though `null` values are not permitted for that attribute in a named Profile.

The reason for the distinction between management clients and SLEE components are as follows:

- Management clients may be driven by human administrators, who are potentially error prone.
- Verification of state may be a non-trivial compute task. SLEE components execute in the critical path and therefore are highly performance sensitive. SLEE components are also more likely to be correct than human users.

Note: In SLEE 1.0 the `profileVerify` method was invoked on a Profile object when the default Profile of a Profile Table was created, in order to verify the state of the default Profile. In SLEE 1.1 this is no longer the case – the `profileVerify` method is not *invoked during the creation of the default Profile. A management client that creates a Profile is expected to populate the Profile with valid state before attempting to commit the Profile creation.*

10.13.2 Profile object lifecycle

A Profile object can be in one of the following three states (see Figure 15).

- Does Not Exist state.
  The Profile object does not exist. It may not have been created or it may have been deleted.
- Pooled state.
  The Profile object exists but is not assigned to any particular Profile.
- Ready state.
  The Profile object is assigned to a Profile. It is ready to receive method invocations through its Profile Local interface or Profile Management interface, and various lifecycle callback method invocations.

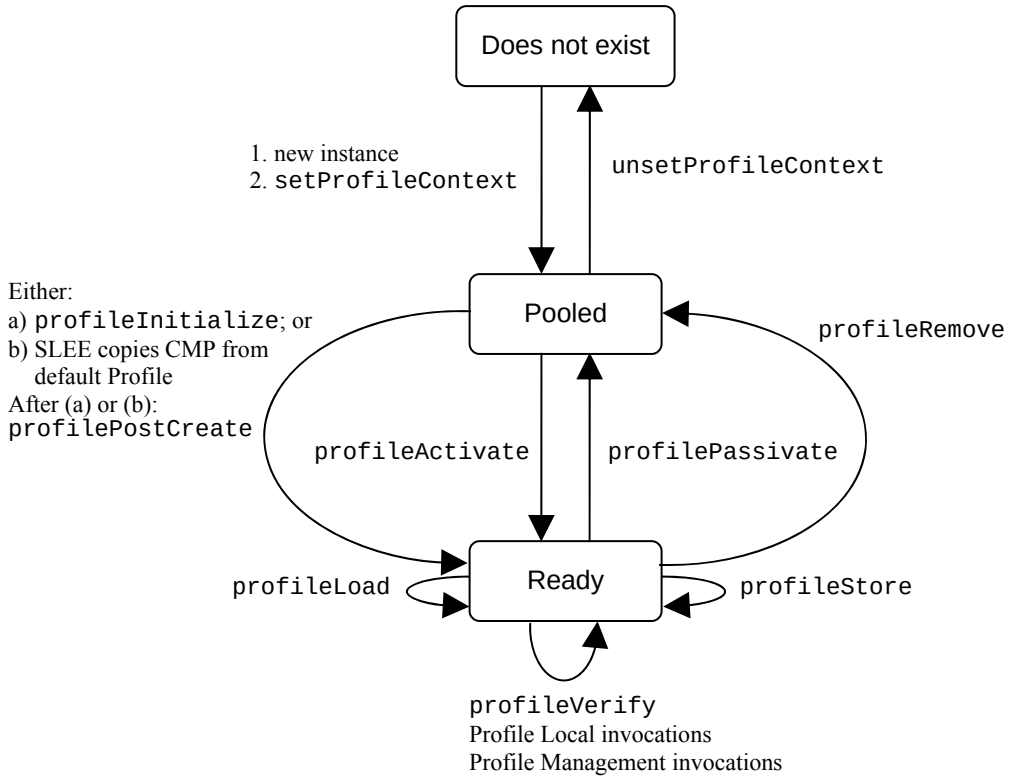

Figure 15 Lifecycle of a Profile object
All specified methods are on the `Profile` interface

The following steps describe the life cycle of a Profile object:

- A Profile object's life cycle starts when the SLEE creates the object using its constructor. The SLEE then invokes the `setProfileContext` method to pass the Profile object a `ProfileContext` object. The `ProfileContext` object allows the Profile object to access functions provided by the SLEE (see Section 10.14).
- The Profile object enters the pool of available Profile objects. Each Profile Table has its own pool of Profile objects. While the Profile object is in the available pool, the Profile object is not associated with any particular Profile. All Profile objects in the pool are considered equivalent
- A Profile object transitions from the Pooled state to the Ready state when the SLEE selects that Profile object to service a Profile Local or Profile Management object invocation. There are three possible transitions from the Pooled state to the Ready state: through the `profileInitialize` and `profilePostCreate` methods, through the `profilePostCreate` method only, or through the `profileActivate` method. The three transitions occur under the following conditions:
    1. The transition through `profileInitialize` and `profilePostCreate` occurs only when the default Profile is created. The `profileInitialize` method sets the CMP state for the default Profile.
    2. The transition through `profilePostCreate` only occurs when any Profile other than the default Profile is created by an invocation of the `create` method on the `ProfileTable` interface, or via the `createProfile` method on the `ProfileProvisioningMBean`. Immediately before the invocation of `profilePostCreate` the SLEE copies the CMP state of the default profile to the CMP state of the new Profile.
    3. The transition through `profileActivate` occurs when the SLEE needs to transition a Profile object from the Pooled state to the Ready state for a Profile which already exists. This occurs, for example, when there is no existing Profile object in the Ready state assigned to the Profile to receive a method invocation for the Profile.

  When a Profile object is in the Ready state, the Profile object is associated with a specific Profile. While the Profile object is in the Ready state, the SLEE can synchronize the transient state held in the Profile object with the persistent state of the Profile whenever it determines the need to, by invoking the `profileLoad` and `profileStore` methods zero or more times. Profile Local and Profile Management invocations can be invoked on the Profile object zero or more times. Invocations of the `profileLoad` and `profileStore` methods can be arbitrarily mixed with invocations of these methods subject to the Profile object lifecycle.

  The SLEE can choose to passivate a Profile object if it wants to disassociate the Profile object from the Profile it is currently associated with, for example to reuse the Profile object for a different Profile. To passivate a Profile object, the SLEE first invokes the `profileStore` method to allow the Profile object to prepare itself for the synchronization of the Profile's persistent state with the Profile object's transient state, and then the SLEE invokes the `profilePassivate` method to return the Profile object to the Pooled state.

  Eventually, the SLEE will transition the Profile object to the Pooled state. There are three possible normal transitions from the Ready state to the Pooled state: through the `profilePassivate` method, through the `profileRemove` method, and because of a transaction rollback for `profilePostCreate` (not shown in above Profile object life cycle state diagram). The SLEE invokes the `profilePassivate` method when the SLEE wants to disassociate the Profile object from the Profile without removing the Profile. The SLEE invokes the `profileRemove` method when the SLEE wants to remove the Profile (i.e. when the Profile is removed due to a `ProfileTable` interface `remove` method invocation or the `ProfileProvisioningMBean` interface `removeProfile` method invocation). If `profilePostCreate` is called and success-

fully returns without throwing a `RuntimeException` or `CreateException`, and subsequently the enclosing transaction rolls back, the SLEE may transition the Profile object back to the Pooled state by invoking the `profilePassivate` method.

- When the Profile object returns to the Pooled state, it is no longer associated with the Profile. The SLEE can assign the Profile object to any Profile of the same Profile Table.

  The SLEE may release its references to a Profile object in the Pooled state, allowing it to be garbage collected, after calling the `unsetProfileContext` method on the Profile object.

Notes:

1. The `ProfileContext` object passed by the SLEE to the Profile object in the `setProfileContext` method is not an object that contains static information. For example, the result of the `getProfileName` method might be different each time a Profile object moves from the Pooled state to the Ready state.
2. A Profile object is only ever associated with one Profile Table. This means the `getProfileTableName` method of a `ProfileContext` object always returns the same result during the lifetime of the Profile object.
3. A `RuntimeException` thrown from any method of a Profile object (including the Profile Local methods, the Profile Management interface methods, and the life cycle callbacks invoked by the SLEE) results in the transition of the Profile object to the Does Not Exist state. The SLEE will not invoke any method on a Profile object after a `RuntimeException` has been thrown from the Profile object. The corresponding Profile continues to exist if it existed before the invocation. The Profile can continue to process invocations because the SLEE can use a different Profile object for the Profile.
4. A SLEE implementation is not required to maintain a pool of Profile objects in the Pooled state. The pooling approach is an example of a possible implementation, but it is not the required implementation. Whether the SLEE implementation uses a pool or not has no bearing on the Profile abstract class coding style.

10.14 `ProfileContext` interface

The SLEE provides each Profile object with a `ProfileContext` object. The `ProfileContext` object gives the Profile object access to the Profile object's context maintained by the SLEE, allows the Profile object to invoke functions provided by the SLEE, and obtain information about the Profile assigned to the Profile object.

A `ProfileContext` object is associated with one Profile Table and the associated Profile Table does not change during the lifetime of that `ProfileContext` object.

The `ProfileContext` object implements the `javax.slee.profile.ProfileContext` interface. The `ProfileContext` interface is as follows:

```
package javax.slee.profile;

import javax.slee.facilities.Tracer;
import javax.slee.TransactionRequiredLocalException;
import javax.slee.SLEEException;

public interface ProfileContext {
       public String getProfileTableName()
              throws  SLEEException;
       public String getProfileName()
              throws  IllegalStateException,
                      SLEEException;
       public ProfileTable getProfileTable()
              throws  SLEEException;
       public ProfileTable getProfileTable(String profileTableName)
              throws  NullPointerException,
                      UnrecognizedProfileTableNameException,
```

```
                        SLEEException;
        public ProfileLocalObject getProfileLocalObject()
                throws  IllegalStateException,
                        SLEEException;
        public Tracer getTracer(String tracerName)
                throws  NullPointerException,
                        IllegalArgumentException,
                        SLEEException;
        public boolean getRollbackOnly()
                throws  TransactionRequiredLocalException,
                        SLEEException;
        public void setRollbackOnly()
                throws  TransactionRequiredLocalException,
                        SLEEException;
}
```

10.14.1 `getProfileTableName` method

This method returns the name of the Profile Table which the Profile Context is associated with.

- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

This method is a non-transactional method.

10.14.2 `getProfileName` method

This method returns the name of the Profile which this Profile object is currently assigned to.

- This method may only be invoked during a `profilePostCreate` method invocation, or during any method invocation on a Profile object while the Profile object is in the Ready state. Otherwise a `java.lang.IllegalStateException` is thrown.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

This method is a non-transactional method

10.14.3 `getProfileTable()` method

This method returns a `ProfileTable` object for the Profile Table which the Profile Context is associated with. The return value implements the Profile Table interface for the associated Profile Table. If the Profile Specification of the Profile Table has defined a Profile Table interface that extends `javax.slee.profile.ProfileTable` then the object returned from this method may be safely typecast to the subinterface declared in the Profile Specification.

- This method throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

This method is a non-transactional method.

10.14.4 `getProfileTable(String profileTableName)` method

This method returns a `ProfileTable` object for the Profile Table with the profile table name specified by the `profileTableName` argument. The return value implements the Profile Table interface of the specified Profile Table. If the Profile Specification of the Profile Table has defined a Profile Table interface that extends `javax.slee.profile.ProfileTable` then the object returned from this method may be safely typecast to the subinterface declared in the Profile Specification. However, if the Profile Specification of the Profile who the `ProfileContext` object belongs to has not declared a `profile-spec-ref` in its deployment descriptor for the Profile Specification of the Profile Table specified by the `profileTableName` argument, then the Profile may not have the necessary classes in its classloader to perform this typecast. In addition, for the same reasons, the Profile may not be able to interact with Profile Local

Objects obtained from the returned `ProfileTable` object other than via the generic SLEE-defined `ProfileLocalInterface`.

- This method throws a `java.lang.NullPointerException` if the `profileTableName` argument is `null`.
- This method throws a `javax.slee.profile.UnrecognizedProfileTableNameException` if no Profile Table exists with the name specified by the `profileTableName` argument.
- This method throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

This method is a non-transactional method.

10.14.5 `getProfileLocalObject` method

This method returns a Profile local object that represents the Profile assigned to the Profile object for the `ProfileContext` object. The object returned is an object that implements the Profile Local interface of the Profile. The returned object may be typecast to the Profile Local interface specified by the Profile Specification of the Profile.

- This method may only be invoked during a `profilePostCreate` method invocation, or during any method invocation on a Profile object while the Profile object is in the Ready state. Otherwise a `java.lang.IllegalStateException` is thrown.
- This method cannot be used to obtain a Profile Local object for the default Profile. Attempting to invoke this method from a Profile object associated with the default Profile causes a `java.lang.IllegalStateException` to be thrown.
- This method throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

This method is a non-transactional method.

10.14.6 `getTracer` method

This method returns a tracer for the specified tracer name. The notification source used by the tracer is a `javax.slee.management.ProfileTableNotification` that contains the name of the Profile Table the Profile Context is associated with. Refer to Section 13.3 for a complete discussion on tracers and tracer names. Trace notifications generated by a tracer obtained using this method are of the type `javax.slee.management.ProfileTableNotification.TRACE_NOTIFICATION_TYPE`.

- This method throws a `java.lang.NullPointerException` if the `tracerName` argument is `null`.
- This method throws a `java.lang.IllegalArgumentException` if the `tracerName` argument is not a valid tracer name. See section 13.3 for valid tracer names.
- This method throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

This method is a non-transactional method.

10.14.7 `getRollbackOnly` method

Test if the current transaction has been marked for rollback. A Profile object invokes this method while executing within a transaction to determine if the transaction has been marked for rollback.

- This method returns `true` if the current transaction has been marked for rollback and `false` otherwise.

- It throws a `javax.slee.TransactionRequiredLocalException` if there is no valid transaction context associated with the calling thread.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

This method is a mandatory transactional method.

10.14.8 `setRollbackOnly` method

Mark the current transaction for rollback. Once a transaction is marked for rollback it can never commit. A Profile object invokes this method when it does not want the current transaction to commit.

- It throws a `javax.slee.TransactionRequiredLocalException` if there is no valid transaction context associated with the calling thread.
- It throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

This method is a mandatory transactional method.

10.15 Hiding and renaming Profile CMP interface accessor methods

The Profile Specification Developer can use the Profile Management interface and/or the Profile Local interface to hide and rename CMP field accessor methods declared in the Profile CMP interface from the Management and SLEE Component views respectively

- CMP field accessor method hiding.
  The Profile Specification Developer hides a CMP field accessor method by declaring a Profile Management interface for management clients or a Profile Local interface for SLEE Components that does not extend the Profile CMP interface and only includes in the Profile Management interface the CMP field accessor methods that should be visible.
- CMP field accessor method renaming.
  Accessor method renaming builds on CMP field accessor method hiding. In addition to hiding the CMP field accessor method, the Profile Specification Developer provides a substitute method with a different name. The Profile Specification Developer provides the implementation of the substitute method in the Profile abstract class. Typically, the substitute method delegates to the hidden CMP field accessor method. (The CMP field accessor method is hidden from management clients and/or SLEE Components but remains public in the Profile abstract class.)

If only the get accessor method of an attribute is visible (either as result of CMP field accessor method hiding or declaring new management or business methods that conform to the attribute get accessor pattern), then the attribute would become visible as a read-only attribute. If only the set accessor method of an attribute is visible, then the attribute would become write-only.

10.15.1 Uses

For example, the Profile Specification Developer may hide the CMP field accessor methods of a date attribute and define alternate methods in the Profile Management interface to read and modify the date attribute. For example, the date attribute may store a date as the number of days since January 1st, 1970. This date format is not human Administrator friendly. The Profile Specification Developer can hide the CMP field accessor methods of this attribute from the Profile MBean interface. It can define a set of new accessor methods in the Profile Management interface with date formats that are more human Administrator friendly, such as “MM/DD/YYYY”.

10.16 A Profile Specification example

The Profile CMP interface of an example Foo Profile Specification is as follows:

```
package com.foobar;
```

```
import javax.slee.profile.ProfileID;

public interface FooProfileCMP {
        public long getDate();
        public void setDate(long date);
        // accessor methods to a QOS Profile
        // in a separate QOS Profile Table
        public ProfileID getQOSProfile();
        public void setQOSProfile(ProfileID profileID);
}
```

The Profile Local interface of this Profile Specification is as follows:

```
package com.foobar;

import javax.slee.profile.ProfileLocalObject;

public interface FooProfileLocal extends ProfileLocalObject {
        // business method which is a substitute for getDate in FooProfileCMP
        public long getDateAsUTC();
}
```

The Profile Management interface of this Profile Specification is as follows:

```
package com.foobar;

import javax.slee.profile.ProfileID;

public interface FooProfileManagement {
        // accessor methods declared in FooProfileCMP
        public ProfileID getQOSProfile();
        public void setQOSProfile(ProfileID profileID);
        // substitute for getDate and setDate in FooProfileCMP
        public String getMMDDYYDate();
        public void setMMDDYYDate(String date);
}
```

The Profile abstract class of this Profile Specification is as follows:

```
package com.foobar;

import javax.slee.CreateException;
import javax.slee.profile.Profile;
import javax.slee.profile.ProfileContext;
import javax.slee.profile.ProfileVerificationException;

public abstract class FooProfileImpl
        implements Profile, FooProfileCMP, FooProfileManagement
{
        public void setProfileContext(ProfileContext ctx) { … }
        public void unsetProfileContext() { … }
        public void profileInitialize() { … }
        public void profilePostCreate() throws CreateException { … }
        public void profileRemove() { … }
        public void profileActivate() { … }
        public void profilePassivate() { … }
        public void profileLoad() { … }
        public void profileStore() { … }
        public void profileVerify() throws ProfileVerificationException { … }

        // implementation of Profile Local methods and Profile Management interface
        public long getDateAsUTC() { … getDate() … }
        public String getMMDDYYDate() { … getDate() … }
        public void setMMDDYYDate(String date) { … setDate(…) … }
}
```

The `profile-spec` element that describes this Profile Specification is as follows:

```
<profile-spec>
        …
        <profile-spec-name> FooProfileSpec </profile-spec-name>
```

```
        <profile-spec-vendor> com.foobar </profile-spec-vendor>
        <profile-spec-version> 3.2.1 </profile-spec-version>
        <profile-classes>
                <profile-cmp-interface >
                        <description> … </description>
                        <profile-cmp-interface-name>
                                com.foobar.FooProfileCMP
                        </profile-cmp-interface-name>
                </profile-cmp-interface>
                <profile-local-interface>
                        <description> … </description>
                        <profile-local-interface-name>
                                com.foobar.FooProfileLocal
                        </profile-local-interface-name>
                </profile-local-interface>
                <profile-management-interface>
                        <description> … </description>
                        <profile-management-interface-name>
                                com.foobar.FooProfileManagement
                        </profile-management-interface-name>
                </profile-management-interface>
                <profile-abstract-class>
                        <description> … </description>
                        <profile-abstract-class-name>
                                com.foobar.FooProfileImpl
                        </profile-abstract-class-name>
                </profile-abstract-class>
        </profile-classes>
        …
</profile-spec>
```

10.17 Java type usage

The following Java type restrictions apply to methods defined in the Profile CMP interface:

- The Java types of the Profile attributes defined in the Profile CMP interface are restricted to Java primitive types and Java serializable types.
- If a Profile Specification defines a Profile Management interface, then the Java types of any Profile attributes who CMP accessor methods are exposed in the Profile Management interface must also be legal RMI-IIOP types.
- If a Profile Specification does not define a Profile Management interface, then the Java types of all Profile attributes must also be legal RMI-IIOP types, as the Profile CMP interface is used as the Profile Management interface in this case.

The following Java type restrictions apply to methods defined in the Profile Management interface:

- The method parameters and return types of all methods exposed to management clients in the Profile Management interface must be legal RMI-IIOP types.

  For easy management using even the most basic of management clients or human Administrators, it is also recommended that the method parameters and return types used in the methods of the Profile Management interface be restricted to the following:

  - Java primitive types, or the equivalent object wrappers
  - `java.lang.String`
  - One-dimensional arrays of the above types

  Any other legal RMI-IIOP type may be used with the caveat that a management client may not be able to manipulate an attribute or invoke a method that uses them. For complex or composite types it is suggested that they be decomposed into their primitive components and each primitive component exposed as separate manageable attribute in the management view, or a management method in the appropriate Profile Management interface be provided that takes the primitive com-

ponents as arguments, composes the composite object, and sets the CMP field with the composite object value.

The follow Java type restrictions apply to methods defined in the Profile Local interface:

- No Java type restrictions apply to methods defined in the Profile Local interface, as all methods have pass-by-reference semantics.

10.18 Method name restrictions

Non-private (such as public, protected, or package private) methods that are declared by the Profile Specification Developer have the following restrictions:

- The method name must not begin with "profile" or "ejb". A SLEE implementation can use method names that begin with "profile" when needed without being concerned with possible method name conflicts with Profile Specification Developer declared method names.
  This restriction does not apply when the Profile Specification Developer implements a "`profile`<XXX>" method declared by the SLEE, such as the life cycle methods declared in the `javax.profile.Profile` interface.
- The method signature must not be the same as any method defined in the `javax.slee.profile.ProfileMBean`, `javax.slee.profile.ProfileManagement`, or `javax.slee.profile.Profile`, unless it is a method in one of these interfaces being implemented.
- The method name should not be one of the method names defined in JMX interfaces such as `javax.management.DynamicMBean` and `javax.management.MBeanRegistration`.

10.19 Profile Specification component environment

The component environment of a Profile Specification component provides the runtime environment that is common to all instances of the Profile abstract class of the Profile Specification component. The Profile Specification component environment is a mechanism that provides the following features:

- Access to some Facilities defined by the SLEE specification.
- Customization of the Profile Specification component during assembly or deployment using deployment descriptor elements.
  This allows a Profile Specification Developer and a Service Deployer or Resource Adaptor Deployer to customize a Profile Specification component's behavior without the need to access or change the Profile Specification component's source code. The SLEE specification defines Profile Specification deployment descriptor elements that add new context information to the component environment of the Profile Specification component. The Profile Specification Developer and the Service Deployer or Resource Adaptor Deployer may use these deployment descriptor elements to provide configuration data that can be determined and fixed during Profile Specification deployment.

10.19.1 Profile Specification component environment as a JNDI naming context

The SLEE implements the Profile Specification component environment, and provides it to the instances of the Profile abstract class through the JNDI interfaces. The Profile Specification component environment is used as follows:

- The methods implemented by the Profile abstract class access the component environment using the JNDI interfaces.
- The SLEE provides an implementation of the JNDI naming context that stores the Profile Specification component environment. The SLEE makes the environment naming context

available to the instances of the Profile abstract class at runtime. These instances use the JNDI interfaces to obtain the values of the environment entries.

- The SLEE provides the following environment entries:
  - An entry for each SLEE Facility exposed to the Profile abstract class.
- The Profile Specification Developer declares in the Profile Specification deployment descriptor all the environment entries that the Profile Specification component expects Administrators to customize during deployment.

Each Profile Specification component has its own set of environment entries. All instances of the Profile abstract class of the same Profile Specification component share the same environment entries; the environment entries are not shared with other Profile Specification components. Instances of the classes of the Profile Specification component are not allowed to modify the Profile Specification component environment at runtime.

Terminology warning: The Profile Specification's component "environment" should not be confused with the "environment properties" defined in the JNDI documentation.

10.19.2 Standard Profile Specification component environment JNDI entries

The SLEE makes available a subset of the SLEE Facilities in the Profile Specification component environment. The following table lists the SLEE specification defined names of the objects of SLEE Facilities under `java:comp/env` that are made available in the Profile Specification component environment.

Objects	*Names*
`AlarmFacility` object	slee/facilities/alarm

10.19.3 Access to Profile component environment

Profile objects locate the environment naming context using the JNDI interfaces. A Profile object creates a `javax.naming.InitialContext` object by using the constructor with no arguments, and looks up the environment naming via the `InitialContext` under the name `java:comp/env`. The Profile Specification component environment entries are stored directly in the environment naming context, or in any of its direct or indirect subcontexts.

The following table lists the names of the SLEE specification defined environment entries that are available to Profiles under `java:comp/env`.

Objects	*Returned Java type*	*Names*
SLEE Facilities	Depends on Facility	slee/facilities/...

For the environment entries declared by Profile Specification Developers, the value of an environment entry is of the Java type declared by the Profile Specification Developer in the Profile deployment descriptor.

The following example illustrates how environment entries are accessed.

```
package com.foobar;

import javax.slee.profile.Profile;
import javax.slee.profile.ProfileContext;
import javax.slee.profile.ProfileTable;
import javax.slee.profile.ProfileVerificationException;
…

public abstract class FooProfileImpl
        implements Profile, FooProfileCMP, FooProfileManagement
{
        …
        public void performFoo(…) {
                …
```

```
                // Obtain the Profile Specification component environment naming context.
                Context initCtx = new InitialContext();
                Context myEnv = (Context)initCtx.lookup("java:comp/env");

                // Obtain the name of the dependent profile table
                // configured by the Service or RA Deployer.
                String profileTableName = (String) myEnv.lookup("tableName");

                // get the ProfileTable object for that table
                ProfileTable table = context.getProfileTable(profileTableName);
                …

                // Get some more environment entries. These environment
                // entries are stored in subcontexts.
                String val1 = (String) myEnv.lookup("foo/name1");
                Boolean val2 = (Boolean) myEnv.lookup("foo/bar/name2");

                // Can also lookup using full pathnames.
                Integer val3 = (Integer)
                initCtx.lookup("java:comp/env/name3");
                Integer val4 = (Integer)
                initCtx.lookup("java:comp/env/foo/name4");
                …
        }
        …
}
```

10.19.4 Declaration of environment entries

The Profile Specification Developer must declare all the environment entries accessed from the Profile abstract class' code. The environment entries are declared using the `env-entry` elements in the Profile Specification deployment descriptor.

Each `env-entry` element describes a single environment entry. The `env-entry` element consists of the following sub-elements:

- An optional `description` element that describes the environment entry.
- An `env-entry-name` element that specifies a name relative to the `java:comp/env` context.
- An `env-entry-type` element that specifies the expected Java type of the environment entry value, i.e. the type of the object returned from the JNDI `lookup` method. The permissible Java types are: `String`, `Character`, `Integer`, `Boolean`, `Double`, `Byte`, `Short`, `Long`, and `Float`.
- An optional `env-entry-value` element that specifies the value of the environment entry.
- An environment entry is scoped to the Profile Specification component whose declaration contains the `env-entry` element. This means that the environment entry is inaccessible from other Profile Specification components at runtime, and that other Profile Specification components may define `env-entry` elements with the same `env-entry-name` without causing a name conflict.
- If the Profile Specification Developer provides a value for an environment entry using the `env-entry-value` element, the value can be changed later by the SBB Developer, Resource Adaptor Developer or Administrator. The value must be a string that is valid for the constructor of the specified type that takes a single `String` parameter, or for `java.lang.Character`, a single character.

The following example is the declaration of environment entries used by the Foo Profile whose code was illustrated in the previous subsection.

```
<profile-spec>
        …
        <profile-classes>
```

```
                …
                <profile-abstract-class>
                        …
                        <profile-abstract-class-name>
                                com.foobar.FooProfileImpl
                        </profile-abstract-class-name>
                </profile-abstract-class>
                …
        </profile-classes>
        …
        <env-entry>
                <description>
                        The name of the dependent profile table.
                </description>
                <env-entry-name> tableName </env-entry-name>
                <env-entry-type> java.lang.String </env-entry-type>
                <env-entry-value>BarTable</env-entry-value>
        </env-entry>
        <env-entry>
                <env-entry-name> foo/name1 </env-entry-name>
                <env-entry-type> java.lang.String </env-entry-type>
                <env-entry-value>value1</env-entry-value>
        </env-entry>
        <env-entry>
                <env-entry-name> foo/bar/name2 </env-entry-name>
                <env-entry-type> java.lang.Boolean </env-entry-type>
                <env-entry-value>true</env-entry-value>
        </env-entry>
        <env-entry>
                <description> Some description. </description>
                <env-entry-name> name3 </env-entry-name>
                <env-entry-type>java.lang.Integer</env-entry-type>
        </env-entry>
        <env-entry>
                <env-entry-name> foo/name4 </env-entry-name>
                <env-entry-type> java.lang.Integer </env-entry-type>
                <env-entry-value>10</env-entry-value>
        </env-entry>
        …
</profile-spec>
```

10.19.4.1 The Profile Specification Deployer's responsibility

The Profile Specification Deployer must ensure that the values of all the environment entries declared by the Profile Specification component are set to meaningful values. The Profile Specification Deployer can modify the values of the environment entries that have been previously set by the Profile Specification Developer, and must set the values of those environment entries for which no value has been specified.

The `description` elements provided by the Profile Specification Developer help the Profile Specification Deployer with this task.

10.20 Profile queries

The SLEE specification defines static queries for management clients and SLEE components (such as SBB's and Resource Adaptors) and dynamic queries for management clients only. Methods defined on the Profile Table interface allow SLEE components to execute static queries (see Section 10.8.2). Methods defined on the `ProfileProvisioningMBean` interface allow management clients to execute static and dynamic queries (see Section 14.10). A future release of this specification may allow SLEE Components to perform dynamic queries in addition to static queries.

10.20.1 Query operators

The following query operators are supported :

- equals.

- not equals.
- less than.
- less than or equals.
- greater than.
- greater than or equals.
- longest prefix match.
- has prefix.
- range match.
- logical and.
- logical or.
- logical not.

The following table shows arguments that apply to query expressions constructed using these query operators.

Query operator	***Query expressions***
equals not equals	These operators compare a Profile attribute to a value. A query expression constructed using one of these operators has two or three arguments. The first argument identifies a Profile attribute. The second argument provides a value that will be compared with the Profile attribute. The Java type of the Profile attribute must be a Java primitive type or its equivalent object wrapper class, `java.lang.String`, or `javax.slee.Address`. Only if the Java type is `java.lang.String`, a third optional argument may be provided. This third argument identifies a collator used for the string comparisons.
less than less than or equals greater than greater than or equals	These operators compare a Profile attribute to a value. A query expression constructed using one of these operators has two or three arguments. The first argument identifies a Profile attribute. The second argument provides a value that will be compared with the Profile attribute. The Java type of the Profile attribute must be a Java primitive type or its equivalent object wrapper class, or `java.lang.String`. Only if the Java type is `java.lang.String`, a third optional argument may be provided. This third argument identifies a collator used for the string comparisons.
longest prefix match	This operator matches with the Profile whose Profile attribute value is the longest prefix match for the value provided as the argument to this operator. Refer to Section 10.20.3.14 for a detailed explanation of the meaning of this operator. This query can only match with Profiles whose Profile attribute value has a character length less than or equal to the length of the value being compared with. Otherwise, the Profile attribute cannot be a prefix of the value – a prefix must be less than or equal to in length of the value it is being compared to. A query expression constructed using this operator has two or three arguments. The first argument of the query expression identifies a Profile attribute. The second argument of the query expression provides a value that the Profile attribute will be compared with. The

	Java type of the Profile attribute must be `java.lang.String`. The optional third argument of the query expression may be provided to identify a collator used for the string comparisons.
has prefix	This operator compares a Profile attribute to a value. A Profile matches if the value is a prefix of the Profile attribute value. This query can only match with Profiles whose Profile attribute value has a character length greater than or equal to the length of the value being compared with. Otherwise the value cannot be a prefix of the Profile attribute – a prefix must be less than or equal to in length of the value it is being compared to. A query expression constructed using this operator has two or three arguments. The first argument of the query expression identifies a Profile attribute. The second argument of the query expression provides a value that the Profile attribute will be compared with. The Java type of the Profile attribute must be `java.lang.String`. The optional third argument of the query expression may be provided to identify a collator used for the string comparisons.
range match	This operator compares a Profile attribute to a range. A range is defined by a "from" value and a "to" value. The comparison returns true if the value of the Profile attribute is greater than or equal to the "from" value and is less than or equal to the "to" value. A query expression constructed using this operator has three or four arguments. The first argument of the query expression identifies a Profile attribute. The second argument of the query expression specifies the "from" value of the range and the third argument specifies the "to" value of the range. The Java type of the Profile attribute must be a Java primitive type or its equivalent object wrapper class, or `java.lang.String`. Only if the Java type is `java.lang.String`, a fourth optional argument may be provided. This fourth argument identifies a collator used for the string comparisons.
and or	These operators are logical operators that operate on the results of other query expressions. A query expression constructed using this operator has at least two arguments. Each of these arguments is a query expression.
not	This operator is a logical operator that operates on the result of another query expression. A query expression constructed using this operator has one argument which is another query expression.

A collator specified in a query expression is used to determine the valid matches when the query is executed. It has no effect on the ordering of results returned.

10.20.2 Static queries

Static queries are defined in a Profile Specification's deployment descriptor. The query expression is known at deployment time and the SLEE carries out the query compilation at deployment time.

Each static query is declared with a `query` element within the `profile-spec` element of the Profile Specification's deployment descriptor.

Each `query` element contains the following attributes and elements:

- A `name` attribute.
  This attribute identifies the query within the Profile Specification. This name is used as part of the query method signature that is declared in the Profile Table interface that is used to invoke this query. Therefore the name element has the following restrictions:
  - It must be a valid Java identifier
- A `description` element.
  This is an optional informational element.
- An optional `query-options` element.
  This element allows options specific to the query to be specified. This element contains the following attributes:
  - A `read-only` attribute.
    This attribute indicates to the SLEE whether a SLEE component executing this query will interact with the Profiles obtained via the query in a read-only or read-write manner within the same transaction the query is executed. This is akin to SQL's `SELECT ...` versus `SELECT ... FOR UPDATE`. If pessimistic locking is being used by the SLEE for the Profile Table, a read-only query allows concurrent access to the same Profiles via a shared reader lock, whereas a read-write query would require the SLEE to obtain an exclusive writer lock on the profile data, prohibiting concurrent access by SLEE components executing in different threads.

    If this option is not specified for a query, the default value is equal to the value of the `profile-read-only` attribute of the enclosing `profile-spec` element.

    It is a deployment error to specify `false` for this element if the value of the `profile-read-only` attribute of the enclosing `profile-spec` element is `true`.
  - A `max-matches` attribute.
    This attribute specifies the maximum number of matching Profiles that can be returned by the SLEE when the query is executed. This option allows smaller manageable sets of data to be returned from a query if it is expected that the query may return a large number of results.

    If this option is not specified for a query, the SLEE will return all matching Profiles when the query is executed.
- Zero or more `query-parameter` elements.
  Each of these elements identifies a parameter of the query. The value of the parameter is provided at runtime. Each `query-parameter` elements has the following attributes:
  - A `name` attribute.
    This attribute defines the name of the query parameter.
  - A `type` attribute.
    This attribute defines the type of the query parameter. The type must be a Java primitive type or its equivalent object wrapper class, or `java.lang.String`.
- Exactly one of the following elements. These elements are generically known as the query expression elements.
  - A `compare` element.
    This element constructs a query expression for a binary operator. It has the following attributes:
    - An `attribute-name` attribute.
      This attribute identifies the Profile attribute that the query expression will operate on. The value of this attribute must be the name of a Profile attribute declared in the Profile CMP interface of same `profile-spec` element. The Java type of the identified Profile attribute must be a Java primitive type or its

equivalent object wrapper, or `java.lang.String`. The `equals` and `not-equals` binary operators also allow the attribute type to be `javax.slee.Address`.

- An `op` attribute.
  This attribute identifies the binary operator to apply to the Profile attribute value. It can be one of the following values: “equals”, “not-equals”, “less-than”, “less-than-or-equals”, “greater-than”, or “greater-than-or-equals”. If the Java type of the Profile attribute is `boolean` or `java.lang.Boolean` then only the “equals”, or “not-equals” operator are allowed.
- An optional `value` attribute.
  This attribute identifies a value that will be compared with the Profile attribute identified by the `attribute-name` attribute using the operator specified by the `op` attribute. The value of this attribute must be a string that can be converted to the Java type of identified attribute.
- An optional `parameter` attribute.
  This attribute identifies a query parameter that will provide values at runtime for comparison with the Profile attribute identified by the `attribute-name` attribute. The value of this attribute must be the name of a query parameter defined in a `query-parameter` element in the same `query` element. Furthermore, the identified query parameter must be the same type as the identified Profile attribute that will be compared.
- An optional `collator-ref` attribute.
  This attribute may only be defined if the Java type of the Profile attribute identified by the `attribute-name` attribute is `java.lang.String`. The value of this attribute references a collator that has been aliased in a `collator` element within the same `profile-spec` element (see Section 10.23.1). This collator will determine the collation order used by the query expression operator.

Either the `value` attribute or the `parameter` attribute, but not both, must be specified for this element.

o A longest-`prefix-match` element.
  This element constructs a query expression using the “longest prefix match” query operator. It has the following attributes:

- An `attribute-name` attribute.
  This attribute identifies the Profile attribute that the query expression will operate on. The value of this attribute must be the name of a Profile attribute declared in the Profile CMP interface of the same `profile-spec` element. The Java type of the identified Profile attribute must be `java.lang.String`.
- An optional `value` attribute.
  This attribute identifies a value that will be compared with the Profile attribute identified by the `attribute-name` attribute.
- An optional `parameter` attribute.
  This attribute identifies a query parameter that will provide values at runtime for comparison with the Profile attribute identified by the `attribute-name` attribute. The value of this attribute must be the name of a query parameter defined in a `query-parameter` element in the same `query` element. Furthermore, the identified query parameter must be of the type `java.lang.String`.
- An optional `collator-ref` attribute.
  The value of this attribute references a collator that has been aliased in a `col-`

`lator` element within the same `profile-spec` element (see Section 10.23.1). This collator will determine the collation order used during the prefix matching.

Either the `value` attribute or the `parameter` attribute, but not both, must be specified for this element.

- A `has-prefix` element.
  This element constructs a query expression using the "has prefix" query operator. It has the following attributes:
  - An `attribute-name` attribute.
    This attribute identifies the Profile attribute that the query expression will operate on. The value of this attribute must be the name of a Profile attribute declared in the Profile CMP interface of the same `profile-spec` element. The Java type of the identified Profile attribute must be `java.lang.String`.
  - An optional `value` attribute.
    This attribute identifies a value that will be compared with the Profile attribute identified by the `attribute-name` attribute.
  - A `parameter` attribute.
    This attribute identifies a query parameter that will provide values at runtime for comparison with the Profile attribute identified by the `attribute-name` attribute. The value of this attribute must be the name of a query parameter defined in a `query-parameter` element in the same `query` element. Furthermore, the identified query parameter must be of the type `java.lang.String`.
  - An optional `collator-ref` attribute.
    The value of this attribute references a collator that has been aliased in a `collator` element within the same `profile-spec` element (see Section 10.23.1). This collator will determine the collation order used during the prefix matching.

  Either the `value` attribute or the `parameter` attribute, but not both, must be specified for this element.
- A `range-match` element.
  This element constructs a query expression using the "range match" query operator. It has the following attributes:
  - An `attribute-name` attribute.
    This attribute identifies the Profile attribute that the query expression will operate on. The value of this attribute must be the name of a Profile attribute declared in the Profile CMP interface of the same `profile-spec` element. The Java type of the identified Profile attribute must be a Java primitive type or its equivalent object wrapper, or `java.lang.String`.
  - An optional `from-value` attribute.
    This attribute identifies a value for the lower bound of the range, and will be compared with the Profile attribute identified by the `attribute-name` attribute. The value of this attribute must be a string that can be converted to the Java type of identified attribute.
  - An optional `from-parameter` attribute.
    This attribute identifies a query parameter that will provide values at runtime for the lower bound of the range, and will be compared with the Profile attribute identified by the `attribute-name` attribute. The value of this attribute must be the name of a query parameter defined in a `query-parameter` element in

the same `query` element. Furthermore, the identified query parameter must be the same type as the identified Profile attribute that will be compared.

- An optional `to-value` attribute.
  This attribute identifies a value for the upper bound of the range, and will be compared with the Profile attribute identified by the `attribute-name` attribute. The value of this attribute must be a string that can be converted to the Java type of identified attribute.
- An optional `to-parameter` attribute.
  This attribute identifies a query parameter that will provide values at runtime for the upper bound of the range, and will be compared with the Profile attribute identified by the `attribute-name` attribute. The value of this attribute must be the name of a query parameter defined in a `query-parameter` element in the same `query` element. Furthermore, the identified query parameter must be the same type as the identified Profile attribute that will be compared.
- An optional `collator-ref` attribute.
  This attribute may only be defined if the Java type of the Profile attribute identified by the `attribute-name` attribute is `java.lang.String`. The value of this attribute references a collator that has been aliased in a collator element within the same profile-spec element (see Section 10.23.1). This collator will determine the collation order of the query expression.

Either the `from-value` attribute or the `from-parameter` attribute, but not both, must be specified for this element. Either the `to-value` attribute or the `to-parameter` attribute, but not both, must also be specified.

- An `and` element.
  This element constructs a query expression using the "and" logical query operator. It has the following sub-elements:
  - Two or more query expression elements, i.e. two or more elements selected from the `compare`, `longest-prefix-match`, `has-prefix`, `range-match`, `and`, `or`, or `not` elements.
- An `or` element.
  This element constructs a query expression using the "or" logical query operator. It has the following sub-elements:.
  - Two or more query expression elements, i.e. two or more elements selected from the `compare`, `longest-prefix-match`, `has-prefix`, `range-match`, `and`, `or`, or `not` elements.
- A `not` element.
  This element constructs a query expression using the "not" logical query operator. It has the following sub-element:
  - A query expression element, i.e. a `compare`, `longest-prefix-match`, `has-prefix`, `range-match`, `and`, `or`, or `not` element.

An example `profile-spec` element that describes a selection of static queries is shown below:

```
<profile-spec>
        …
        <collator strength="Primary">
                <collator-alias>primary</collator-alias>
                <locale-language> … </locale-language>
        </collator>
        …
        <!--'attrBaz' equals "42"-->
        <query name="q0">
```

```
                <compare attribute-name="attrBaz" op="equals" value="42"/>
        </query>
        <!--‘attrBar’ equals ‘paramBar’ with collation order “primary”-->
        <query name="q1">
                <query-parameter name="paramBar" type="java.lang.String"/>
                <compare attribute-name="attrBar" op="equals" parameter="paramBar"
                         collator-ref="primary"/>
        </query>
        <!--‘attrBar’ does not equal ‘paramBar’-->
        <query name="q2">
                <query-parameter name="paramBar" type="java.lang.String"/>
                <not>
                        <compare attribute-name="attrBar" op="equals"
                                parameter="paramBar"/>
                </not>
        </query>
        <!--‘attrBar’ does not equal ‘paramBar’ (alternative) -->
        <query name="q2">
                <query-parameter name="paramBar" type="java.lang.String"/>
                <compare attribute-name="attrBar" op="not-equals" parameter="paramBar"/>
        </query>
        <!--‘attrBar’ equals ‘paramBar’ or ‘attrBar’ equals “fooValue”-->
        <query name="q3">
                <query-parameter name="paramBar" type="java.lang.String"/>
                <or>
                        <compare attribute-name="attrBar" op="equals"
                                parameter="paramBar"/>
                        <compare attribute-name="attrBar" op="equals" value="fooValue"/>
                </or>
        </query>
        <!--attrBar’ equals ‘paramBar’ and ‘attrBaz’ is less than “42”
          or ‘attrBar’ equals “fooValue”-->
        <query name="q4">
                <query-parameter name="paramBar" type="java.lang.String"/>
                <or>
                        <and>
                                <compare attribute-name="attrBar" op=”equals"
                                        parameter="paramBar"/>
                                <compare attribute-name="attrBaz" op="less-than"
                                        value="42"/>
                        </and>
                        <compare attribute-name="attrBar" op="equals" value="fooValue"/>
                </or>
        </query>
        <!--‘attrBaz’ is within range from “13” to “42” -->
        <query name="q5">
                <range-match attribute-name="attrBaz" from-value="10" to-value="42"/>
        </query>
        <--‘attrBaz’ is within range specified by ‘from’ and ‘to’ -->
        <query name="q6">
                <query-parameter name="from" type="int"/>
                <query-parameter name="to" type="int"/>
                <range-match attribute-name="attrBaz" from-parameter="from"
                                to-parameter="to"/>
        </query>
        …
</profile-spec>
```

10.20.3 Dynamic Profile queries

Dynamic queries are defined using the concrete sub-classes of the `QueryExpression` class. A dynamic query may be provided to the SLEE at runtime and the SLEE has to be able to perform the query compilation at runtime. The dynamic queries are only available to management clients via the `ProfileProvisioningMBean` interface (see Section 14.10).

10.20.3.1 `QueryExpression` class

The `QueryExpression` class is the abstract base class of all query expression objects.

The public interface of the `QueryExpression` class is as follows:

```
package javax.slee.profile.query;

import java.io.Serializable;

public abstract class QueryExpression implements Serializable {
        public final String toString() { … }
}
```

- The `toString` method.
  This method returns a string representation of the query expression.

10.20.3.2 `SimpleQueryExpression` class

The `SimpleQueryExpression` class is the abstract base class of all query expression objects that operate directly on profile attributes and values. The public interface of the `SimpleQueryExpression` class is as follows:

```
package javax.slee.profile.query;

public abstract class SimpleQueryExpression extends QueryExpression {
        public final String getAttributeName() { … }
        public final Object getAttributeValue() { … }
        public final QueryCollator getCollator() { … }
}
```

- The `getAttributeName` method.
  This method returns the name of the Profile attribute that this query expression will operate on.
- The `getAttributeValue` method.
  This method returns the value that the Profile attribute will be compared to.
- The `getCollator` method.
  This method returns the collator for this query expression, if one was specified. See Section 10.23.2 for a description of the `QueryCollator` class.

10.20.3.3 `OrderedQueryExpression` class

The `OrderedQueryExpression` class extends the `SimpleQueryExpression` class to define the abstract base class of all query expressions that compare a profile attribute value for order. The public interface of the `OrderedQueryExpression` class is as follows:

```
package javax.slee.profile.query;

public abstract class OrderedQueryExpression extends SimpleQueryExpression {
}
```

10.20.3.4 `CompositeQueryExpression` class

The `CompositeQueryExpression` class is the abstract base class of all query expressions that are composed of two or more other query expressions. The public interface of the `CompositeQueryExpression` class is as follows:

```
package javax.slee.profile.query;

public abstract class CompositeQueryExpression extends QueryExpression {
        public final QueryExpression[] getExpressions() { … }
}
```

- The `getExpressions` method.
  This method returns the nested query expressions that this query expression is composed from.

10.20.3.5 `Equals` class

The `Equals` class defines the Java type used to construct a query expression using the "equals" query operator. A Profile will satisfy this query expression if the value of the Profile attribute identified by the `getAttributeName` method of the `Equals` object is identical to the value returned by the `getAttributeValue` method of the `Equals` object. (The `getAttributeName` and `getAttributeValue` methods are inherited from the `SimpleQueryExpression` superclass.)

The public interface of the `Equals` class is as follows.

```
package javax.slee.profile.query;

public final class Equals extends SimpleQueryExpression {
        // constructors
        public Equals(String attrName, Object attrValue)
                throws NullPointerException { … }
        public Equals(String attrName, String attrValue, QueryCollator collator)
                throws NullPointerException { … }
}
```

- The two-argument constructor.
  This constructor creates a new `Equals` object. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument.
- The three-argument constructor.
  This constructor creates a new `Equals` object for a Profile attribute of type `java.lang.String` where a collator for the comparison is specified. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument. The collator to use for the comparison is specified by the `collator` argument.
- The above constructors may throw the following exception:
  - `java.lang.NullPointerException`.
    If either the `attrName` or `attrValue` argument is null, a `NullPointerException` is thrown.

10.20.3.6 `NotEquals` class

The `NotEquals` class defines the Java type used to construct a query expression using the "not equals" query operator. A Profile will satisfy this query expression if the value of the Profile attribute identified by the `getAttributeName` method of the `NotEquals` object is different to the value returned by the `getAttributeValue` method of the `NotEquals` object. (The `getAttributeName` and `getAttributeValue` methods are inherited from the `SimpleQueryExpression` superclass.)

The public interface of the `NotEquals` class is as follows.

```
package javax.slee.profile.query;

public final class NotEquals extends SimpleQueryExpression {
        // constructors
        public NotEquals(String attrName, Object attrValue)
                throws NullPointerException { … }
        public NotEquals(String attrName, String attrValue, QueryCollator collator)
                throws NullPointerException { … }
}
```

- The two-argument constructor.
  This constructor creates a new `NotEquals` object. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument.
- The three-argument constructor.
  This constructor creates a new `NotEquals` object for a Profile attribute of type `java.lang.String` where a collator for the comparison is specified. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument. The collator to use for the comparison is specified by the `collator` argument.
- The above constructors may throw the following exception:
  - `java.lang.NullPointerException`.
    If either the `attrName` or `attrValue` argument is null, a `NullPointerException` is thrown.

10.20.3.7 `LessThan` class

The `LessThan` class defines the Java type used to construct a query expression using the "less than" query operator. A Profile will satisfy this query expression if the value of the Profile attribute identified by the `getAttributeName` argument of the `LessThan` object is smaller than the value returned by the `getAttributeValue` method of the `LessThan` object. (The `getAttributeName` and `getAttributeValue` methods are inherited from the `SimpleQueryExpression` superclass.)

The public interface of the `LessThan` class is as follows.

```
package javax.slee.profile.query;

public final class LessThan extends OrderedQueryExpression {
        // constructors
        public LessThan(String attrName, Object attrValue)
                throws NullPointerException, IllegalArgumentException { … }
        public LessThan(String attrName, String attrValue, QueryCollator collator)
                throws NullPointerException, IllegalArgumentException  { … }
}
```

- The two-argument constructor.
  This constructor creates a new `LessThan` object. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument. The type of the specified Profile attribute must either be a Java primitive type or a class that implements the `java.lang.Comparable` interface.
- The three-argument constructor.
  This constructor creates a new `LessThan` object for a Profile attribute of type `java.lang.String` where a collator for the comparison is specified. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument. The collator to use for the comparison is specified by the `collator` argument.
- The above constructors may throw the following exceptions:
  - `java.lang.NullPointerException`.
    If either the `attrName` or `attrValue` argument is null, a `NullPointerException` is thrown.
  - `java.lang.IllegalArgumentException`.
    If the class of the `attrValue` argument does not implement the `Comparable` interface, an `IllegalArgumentException` is thrown.

10.20.3.8 `LessThanOrEquals` class

The `LessThanOrEquals` class defines the Java type used to construct a query expression using the "less than or equals" query operator. A Profile will satisfy this query expression if the value of the Profile attribute identified by the `getAttributeName` method of the `LessThanOrEquals` object is smaller than or equal to the value returned by the `getAttributeValue` method of the `LessThanOrEquals` object. (The `getAttributeName` and `getAttributeValue` methods are inherited from the `SimpleQueryExpression` superclass.)

The public interface of the `LessThanOrEquals` class is as follows.

```
package javax.slee.profile.query;

public final class LessThanOrEquals extends OrderedQueryExpression {
        // constructors
        public LessThanOrEquals(String attrName, Object attrValue)
                throws NullPointerException, IllegalArgumentException { … }
        public LessThanOrEquals(String attrName, String attrValue, QueryCollator collator)
                throws NullPointerException, IllegalArgumentException { … }
}
```

- The two-argument constructor.
  This constructor creates a new `LessThanOrEquals` object. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument. The type of the specified Profile attribute must either be a Java primitive type or a class that implements the `java.lang.Comparable` interface.
- The three-argument constructor.
  This constructor creates a new `LessThanOrEquals` object for a Profile attribute of type `java.lang.String` where a collator for the comparison is specified. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument. The collator to use for the comparison is specified by the `collator` argument.
- The above constructors may throw the following exceptions:
  - `java.lang.NullPointerException`.
    If either the `attrName` or `attrValue` argument is null, a `NullPointerException` is thrown.
  - `java.lang.IllegalArgumentException`.
    If the class of the `attrValue` argument does not implement the `Comparable` interface, an `IllegalArgumentException` is thrown.

10.20.3.9 `GreaterThan` class

The `GreaterThan` class defines the Java type used to construct a query expression using the "greater than" query operator. A Profile will satisfy this query expression if the value of the Profile attribute identified by the get`AttributeName` argument of the `GreaterThan` object is larger than the value returned by the `getAttributeValue` method of the `GreaterThan` object. (The `getAttributeName` and `getAttributeValue` methods are inherited from the `SimpleQueryExpression` superclass.)

The public interface of the `GreaterThan` class is as follows.

```
package javax.slee.profile.query;

public final class GreaterThan extends OrderedQueryExpression {
        // constructors
        public GreaterThan(String attrName, Object attrValue)
                throws NullPointerException, IllegalArgumentException { … }
        public GreaterThan(String attriName, String attrValue, QueryCollator collator)
                throws NullPointerException, IllegalArgumentException { … }
```

```
}
```

- The two-argument constructor.
  This constructor creates a new `GreaterThan` object. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument. The type of the specified Profile attribute must either be a Java primitive type or a class that implements the `java.lang.Comparable` interface.
- The three-argument constructor.
  This constructor creates a new `GreaterThan` object for a Profile attribute of type `java.lang.String` where a collator for the comparison is specified. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument. The collator to use for the comparison is specified by the `collator` argument.
- The above constructors may throw the following exceptions:
  - `java.lang.NullPointerException`.
    If either the `attrName` or `attrValue` argument is null, a `NullPointerException` is thrown.
  - `java.lang.IllegalArgumentException`.
    If the class of the `attrValue` argument does not implement the `Comparable` interface, an `IllegalArgumentException` is thrown.

10.20.3.10 `GreaterThanOrEquals` class

The `GreaterThanOrEquals` class defines the Java type used to construct a query expression using the "greater than or equals" query operator. A Profile will satisfy this query expression if the value of the Profile attribute identified by the `getAttributeName` method of the `GreaterThanOrEquals` object is larger than or equal to the value returned by the `getAttributeValue` method of the `GreaterThanOrEquals` object. (The `getAttributeName` and `getAttributeValue` methods are inherited from the `SimpleQueryExpression` superclass.)

The public interface of the `GreaterThanOrEquals` class is as follows.

```
package javax.slee.profile.query;

public final class GreaterThanOrEquals extends OrderedQueryExpression {
        // constructors
        public GreaterThanOrEquals(String attrName, Object attrValue)
                throws NullPointerException, IllegalArgumentException { … }
        public GreaterThanOrEquals(String attrName, String attrValue,
                                   QueryCollator collator)
                throws NullPointerException, IllegalArgumentException { … }
}
```

- The two-argument constructor.
  This constructor creates a new `GreaterThanOrEquals` object. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument. The type of the specified Profile attribute must either be a Java primitive type or a class that implements the `java.lang.Comparable` interface.
- The three-argument constructor.
  This constructor creates a new `GreaterThanOrEquals` object for a Profile attribute of type `java.lang.String` where a collator for the comparison is specified. The Profile attribute to compare is specified by the `attrName` argument, and the value to compare the Profile attribute with is specified by the `attrValue` argument. The collator to use for the comparison is specified by the `collator` argument.

- The above constructors may throw the following exceptions:
  - `java.lang.NullPointerException`.
    If either the `attrName` or `attrValue` argument is null, a `NullPointerException` is thrown.
  - `java.lang.IllegalArgumentException`.
    If the class of the `attrValue` argument does not implement the `Comparable` interface, an `IllegalArgumentException` is thrown.

10.20.3.11 And class

The `And` class defines the Java type used to construct a query expression using the "and" query operator. This query expression returns `true` if all of the query expressions in the `And` object return `true`, otherwise it returns `false`.

The public interface of the `And` class is as follows.

```
package javax.slee.profile.query;

public final class And extends CompositeQueryExpression {
        // constructors
        public And(QueryExpression expr1, QueryExpression expr2)
                throws NullPointerException { … };
        public And(QueryExpression[] exprs)
                throws NullPointerException { … }

        // methods
        public And and(QueryExpression expr)
                throws NullPointerException, IllegalArgumentException { … }
}
```

- The two-argument constructor.
  This constructor creates a new `And` object composed of the query expressions specified by the `expr1` and `expr2` arguments.
- The one-argument constructor.
  This constructor creates a new `And` object composed of the query expressions contained in the array specified by the `exprs` argument.
- The `and` method.
  This method adds the query expression specified by the `expr` argument to this `And` query expression object. This method allows an `And` query expression to be composed, for example, as follows:

  ```
  And expr = new And(expr1, expr2).and(expr3).and(expr4)… ;
  ```

  This method throws the following exception:
  - `java.lang.IllegalArgumentException`.
    If adding the expression specified by the `expr` argument to this `And` expression would generate a cyclic expression, and `IllegalArgumentException` is thrown.
- The above constructors and methods may throw the following exception:
  - `java.lang.NullPointerException`.
    If any argument is null, a `NullPointerException` is thrown.

10.20.3.12 Or class

The `Or` class defines the Java type used to construct a query expression using the "or" query operator. This query expression returns `true` if any of the query expressions in the `Or` object returns `true`, otherwise it returns `false`.

The public interface of the `Or` class is as follows.

```
package javax.slee.profile.query;

public final class Or extends CompositeQueryExpression {
        // constructors
        public Or(QueryExpression expr1, QueryExpression expr2)
                throws NullPointerException { … };
        public Or(QueryExpression[] exprs)
                throws NullPointerException { … }

        // methods
        public Or or(QueryExpression expr)
                throws NullPointerException, IllegalArgumentException { … }
}
```

- The two-argument constructor.
  This constructor creates a new `Or` object composed of the query expressions specified by the `expr1` and `expr2` arguments.
- The one-argument constructor.
  This constructor creates a new `Or` object composed of the query expressions contained in the array specified by the `exprs` argument.
- The `or` method.
  This method adds the query expression specified by the `expr` argument to this `Or` query expression object. This method allows an `Or` query expression to be composed, for example, as follows:

  ```
  Or expr = new Or(expr1, expr2).or(expr3).or(expr4)… ;
  ```

  This method throws the following exception:
  - `java.lang.IllegalArgumentException`.
    If adding the expression specified by the `expr` argument to this `Or` expression would generate a cyclic expression, and `IllegalArgumentException` is thrown.
- The above constructors and methods may throw the following exception:
  - `java.lang.NullPointerException`.
    If any argument is null, a `NullPointerException` is thrown.

10.20.3.13 `Not` class

The `Not` class defines the Java type used to construct a query expression using the "not" query operator. This query expression returns `true` if the query expression in the `Not` object returns `false`, and `false` if the query expression in the `Not` object returns `true`.

The public interface of the `Not` class is as follows.

```
package javax.slee.profile.query;

public final class Not extends QueryExpression {
        // constructor
        public Not(QueryExpression expr)
                throws NullPointerException { … }

        // methods
        public QueryExpression getExpression() { … }
}
```

- The constructor.
  The constructor creates a new `Not` object composed of the query expression specified by the `expr` argument.
  This constructor throws the following exception:

  - `java.lang.NullPointerException`.
    If the `expr` argument is null, a `NullPointerException` is thrown.

- The `getExpression` method.
  This method returns the nested query expression that this query expression is composed from.

10.20.3.14 `LongestPrefixMatch` class

The `LongestPrefixMatch` class defines the Java type used to construct a query expression using the "longest prefix match" query operator. A Profile will satisfy the query criteria if the Profile attribute identified by the `getAttributeName` method of the `LongestPrefixMatch` object has a value that is the longest prefix match for the string specified by the `getAttributeValue` method of the `LongestPrefixMatch` object. (The `getAttributeName` and `getAttributeValue` methods are inherited from the `SimpleQueryExpression` superclass.) A longest prefix match occurs when no other prefix values specified by this attribute in the Profiles within the Profile Table matches more characters starting from the first character of the `getAttributeValue` value. It is possible to have more than one Profile satisfy the longest prefix match query if the attribute within the Profile Table is not unique.

As a practical example of how this query expression could be used, consider a Profile Specification that contains an attribute named `prefix` of type `java.lang.String`[21]. Consider a Profile Table that contains Profiles of this Profile Specification as shown below:

Profile name	Value of `prefix` attribute in Profile
A	1
B	12
C	123
D	1234
E	124

The table below indicates which Profiles in this Profile Table would match using the "longest prefix match" query operator when the specified value argument is given to the operator:

Query operator value argument	Matching Profile name
1653333	A
1256999	B
1238764	C
1234567	D
1247123	E
2987654	*none*

Specifically:

- Profile A matches with the value `1653333` as the prefix "`1`" contained in Profile A is the longest prefix match for this value.
- Profile D matches with the value `1234567` as the prefix "`1234`" contained in Profile D is the longest prefix match for this value.

[21] Longest prefix matching can only be performed against Profile attributes of type `java.lang.String`.

- No profile matches with the value `2987654` as no Profile contains the prefix "`2`" (or "`29`", or "`298`", and so on).

The public interface of the `LongestPrefixMatch` class is as follows.

```
package javax.slee.profile.query;

public final class LongestPrefixMatch extends SimpleQueryExpression {
        // constructors
        public LongestPrefixMatch(String attrName, String attrValue)
                throws NullPointerException { … }
        public LongestPrefixMatch(String attrName, String attrValue,
                                  QueryCollator collator)
                throws NullPointerException { … }
}
```

- The two-argument constructor.
  This constructor creates a new `LongestPrefixMatch` object. The Profile attribute to compare is specified by the `attrName` argument. This Profile attribute must be of the type `java.lang.String`. The value to compare the Profile attribute with is specified by the `attrValue` argument.
- The three-argument constructor.
  This constructor creates a new `LongestPrefixMatch` object where a collator for the prefix match comparison is specified. The Profile attribute to compare is specified by the `attrName` argument. This Profile attribute must be of the type `java.lang.String`. The value to compare the Profile attribute with is specified by the `attrValue` argument and the collator to use for the comparison is specified by the `collator` argument.
- The above constructors may throw the following exception:
  - `java.lang.NullPointerException`.
    If either the `attrName` or `attrValue` argument is null, a `NullPointerException` is thrown.

10.20.3.15 `HasPrefix` class

The `HasPrefix` class defines the Java type used to construct a query expression using the "has prefix" query operator. A Profile will satisfy the query criteria if the value specified by the `getAttribute-Value` method of the `HasPrefix` object is a prefix of the profile attribute identified by the `getAt-tributeName` method of the `HasPrefix` object. It is possible to have more than one Profile satisfy the has prefix query if the specified value is a prefix of the attribute value in multiple Profiles.

The public interface of the `HasPrefix` class is as follows.

```
package javax.slee.profile.query;

public final class HasPrefix extends SimpleQueryExpression {
        // constructors
        public HasPrefix(String attrName, String attrValue)
                throws NullPointerException { … }
        public HasPrefix(String attrName, String attrValue,
                                  QueryCollator collator)
                throws NullPointerException { … }
}
```

- The two-argument constructor.
  This constructor creates a new `HasPrefix` object. The Profile attribute to compare is specified by the `attrName` argument. This Profile attribute must be of the type `java.lang.String`. The value to compare the Profile attribute with is specified by the `attrValue` argument.
- The three-argument constructor.
  This constructor creates a new `HasPrefix` object where a collator for the prefix match

comparison is specified. The Profile attribute to compare is specified by the `attrName` argument. This Profile attribute must be of the type `java.lang.String`. The value to compare the Profile attribute with is specified by the `attrValue` argument and the collator to use for the comparison is specified by the `collator` argument.

- The above constructors may throw the following exception:
  - `java.lang.NullPointerException`.
    If either the `attrName` or `attrValue` argument is null, a `NullPointerException` is thrown.

10.20.3.16 `RangeMatch` class

The `RangeMatch` class defines the Java type used to construct a query expression using the "range match" query operator. A Profile will satisfy the query expression if the Profile attribute identified by the `getAttributeName` method of the `RangeMatch` object is greater than or equal to the value returned by the `getFromValue` method of the `RangeMatch` object and less than or equal to the value returned by the `getToValue` method of the `RangeMatch` object.

The public interface of the `RangeMatch` class is as follows.

```
package javax.slee.profile.query;

public final class RangeMatch extends QueryExpression {
        // constructors
        public RangeMatch(String attrName, Object fromValue, Object toValue)
                throws NullPointerException { … }
        public RangeMatch(String attrName, String fromValue, String toValue,
                        QueryCollator collator)
                throws NullPointerException { … }

        // methods
        public String getAttributeName() { … }
        public Object getFromValue() { … }
        public Object getToValue() { … }
}
```

- The three-argument constructor.
  This constructor creates a new `RangeMatch` object. The Profile attribute to compare is specified by the `attrName` argument. The lower bound value of the range to compare the Profile attribute with is specified by the `fromValue` argument and the upper bound value of the range is specified by the `toValue` argument.
- The three-argument constructor.
  This constructor creates a new `RangeMatch` object for a Profile attribute of type `java.lang.String` where a collator for the comparison is specified. The Profile attribute to compare is specified by the `attrName` argument. The lower bound value of the range to compare the Profile attribute with is specified by the `fromValue` argument and the upper bound value of the range is specified by the `toValue` argument. The collator to use for the comparison is specified by the `collator` argument.
- The above constructors may throw the following exception:
  - `java.lang.NullPointerException`.
    If either the `attrName` or `attrValue` argument is null, a `NullPointerException` is thrown.
- The `getAttributeName` method.
  This method returns the name of the Profile attribute that this query expression will operate on.
- The `getFromValue` and `getToValue` methods.
  These methods return the values of the lower bound and upper bound of the range respectively.

10.21 Profile attribute uniqueness

Profile attributes may have a constraint that their values are unique within a Profile Table. Attribute uniqueness is controlled via entries in the Profile Specifications deployment descriptor (see section 3.3.7). Attribute uniqueness may only be specified for CMP fields whose Java type is a primitive type, an Object wrapper of a primitive type (e.g. `java.lang.Integer`), `java.lang.String`, or `javax.slee.Address`.

To declare the uniqueness constraint for a Profile attribute a `cmp-field` element is defined that identifies the attribute, and the `unique` attribute within the element is declared with the value "True". For attributes where the Java type is `java.lang.String` a collator may be specified to define *equality*. Collators are often used for internationalized strings, case insensitive comparisons etc. The collator is specified by the `unique-collator-ref` attribute within the `cmp-field` element.

The following is an example of a Profile CMP interface with a unique attribute.

```
package com.example;

public interface MyProfileCMP{
…

    // an attribute representing an internationally formatted mobile number
    public String getMSISDN();
    public void setMSISDN(String msisdn);

…
}
```

The following is a fragment of a deployment descriptor that declares the attribute as being unique.

```
<profile-spec>
…
    <profile-classes>
        <description>Uniqueness example for the SLEE spec</description>
        <profile-cmp-interface>
            <profile-cmp-interface-name>
                com.example.MyProfileCMP
            </profile-cmp-interface-name>
            <cmp-field unique="True">
                <description>Internationally formatted mobile number</description>
                <cmp-field-name>mSISDN</cmp-field-name>
            </cmp-field>
        </profile-cmp-interface>
    </profile-classes>
…
</profile-spec>
```

10.22 Profile attribute indexing

Changed in 1.1: various types of query were added to Profiles, therefore corresponding indexing hints were added. The SLEE no longer requires a Profile attribute to be indexed in order to retrieve the Profile, as appropriate indexing for Profiles and usage patterns is the responsibility of the Profile Specification Developer and Administrator. Therefore methods requiring a Profile attribute to be indexed were deprecated. Support for collators has been included allowing support for internationalized String attributes.

An index on a table provides a means to rapidly locate information. Properly defining and using indexes improves the general performance of queries.

The `getProfileByAttribute` and `getProfilesByAttribute` methods on the `ProfileProvisioningMBean` and `ProfileFacility` interfaces, and the `findProfileByAttribute` and `findProfilesByAttribute` methods on the `ProfileTable` interface should be used for attributes with appropriate indexing.

Profile attributes may provide indexing hints for use by the SLEE. Indexing hints are are specified via entries in the Profile Specifications deployment descriptor (see section 3.3.7). Indexing hints may only be

specified for CMP fields whose Java type is a primitive type, an Object wrapper of a primitive type (e.g. `java.lang.Integer`), `java.lang.String`, or `javax.slee.Address`.

To declare indexing hints for a Profile attribute a `cmp-field` element is defined that identifies the attribute, and one or more `index-hint` sub-elements. Each index-hint element contains a `query-operator` attribute. This attribute identifies a type of query that may be executed on the Profile attribute.

For attributes where the Java type is `java.lang.String` a collator may be specified to define *equality and ordering*. Collators are often used for internationalized strings, case insensitive comparisons etc. The collator for an indexing hint is specified by the `collator-ref` attribute within the `index-hint` element.

The following is an example of a Profile CMP interface that is used for determining the county for a internationally formatted phone number. In this example an index is defined that is suitable for the longest-prefix-match query operation.

```
package com.example;

public interface CountryPrefixProfileCMP{
…

    // an attribute representing a country code
    public String getCountryCode();
    public void setCountryCode(String code);

    // an attribute specifying the country name
    public String getCountryName();
    public void setCountryName(String name);

…
}
```

A profile table for this profile specification would include multiple entries, one for each country.

The following is a fragment of a deployment descriptor declares country code is unique, and provides an indexing hint for the longest-prefix-match query.

```
<profile-spec>
…
    <profile-classes>
        <description>Indexing example for the SLEE spec</description>
        <profile-cmp-interface>
            <profile-cmp-interface-name>
                com.example.CountryPrefixProfileCMP
            </profile-cmp-interface-name>
            <cmp-field unique=”True”>
                <description>Country Code</description>
                <cmp-field-name>countryCode</cmp-field-name>
                <index-hint query-operator=”longest-prefix-match”/>
            </cmp-field>
        </profile-cmp-interface>
    </profile-classes>
…
</profile-spec>
```

Table 1 shows a hypothetical country code Profile Table for the Profile Specification shown above. In this table there are only five countries. If a `longest-prefix-match` query was executed with the argument “441234567890” the matching profile would be for the U.K. If the same query was executed with the argument “11234567890” the matching profile would be for the U.S.A, and if we executed the same query with the argument “791234567890” no matching profile would be found. For more information on profile queries refer to section 10.20.

Country code	**Country name**
1	U.S.A.

34	Spain
44	U.K.
49	Germany
64	New Zealand

Table 1 Hypothetical county code Profile Table

10.23 Collators

Collators can be used in Profile indexes and queries for locale-sensitive comparisons of Profile attributes of type `java.lang.String`. A collator may be specified as part of a static or dynamic Profile query to alter the set of possible matches that may occur. A collator specified in a query expression is used to determine the valid matches when the query is executed. It has no effect on the ordering of results returned.

The support for collators in Profile queries and indexes in the SLEE specification is based on the `java.text.Collator` class supported by the Java 2 Platform. A collator is primarily identified by the language, country, and variant of the represented locale, and further modified by its strength and decomposition properties.

Collators for profile indexes and static queries are declared in the Profile Specification's deployment descriptor. Collators for dynamic queries are specified in a `QueryExpression` object using the `QueryCollator` class.

10.23.1 Statically defined collators

A statically defined collator is declared in the Profile Specification deployment descriptor by the `collator` element. The collator element has the following attributes and sub-elements:

- The optional `strength` attribute.
  This attribute specifies the collator's strength property. The strength property determines the minimum level of difference considered significant during comparison. If unspecified, the default strength for the specified locale is used.
- The optional `decomposition` attribute.
  This attribute specifies the collator's decomposition mode. The decomposition mode determines how Unicode composed characters are handled. If unspecified, the default decomposition mode for the specified locale is used.
- The `collator-alias` attribute.
  This element specifies an alias for the collator. Other elements in the same `profile-spec` element use this alias to reference the collator. The scope of this alias is the enclosing `profile-spec` element, and each alias must be unique within that scope.
- The `locale-language` element.
  This element specifies an ISO language code that identifies the language of the collator's locale. These codes are the lower-case, two-letter codes as defined by ISO-639.
- The optional `locale-country` element.
  This element specifies an ISO country code that identifies a specific country for the locale language. These codes are the upper-case, two-letter codes as defined by ISO-3166.
- The optional `locale-variant` element.
  This element specifies a vendor or browser-specific variant to a locale language. If this element is specified the `locale-country` element must also be specified.

Some examples of statically defined collators are as follows:

```
<profile-spec>
```

```
        …
        <collator strength="Primary">
                <collator-alias>Spanish_Primary</collator-alias>
                <locale-language>es</locale-language>
        </collator>
        <collator strength="Tertiary">
                <collator-alias>Spanish_Tertiary</collator-alias>
                <locale-language>es</locale-language>
                <locale-country>ES</locale-country>
                <locale-variant>Modern</locale-variant>
        </collator>
        <collator>
                <collator-alias>US_English</collator-alias>
                <locale-language="en"</locale-language>
                <locale-country>US</locale-country>
        </collator>
        <collator>
                <collator-alias>UK_English</collator-alias>
                <locale-language="en"</locale-language>
                <locale-country>UK</locale-country>
        </collator>
        …
</profile-spec>
```

10.23.2 QueryCollator class

The `QueryCollator` class is a serializable class used to describe a collator for a dynamic query expression[22]. The public interface of the `QueryCollator` class is as follows:

```
package javax.slee.profile.query;

import java.io.Serializable;
import java.text.Collator;
import java.util.Locale;

public final class QueryCollator implements Serializable {
        // constructors
        public QueryCollator(Locale locale)
                 throws NullPointerException { … }
        public QueryCollator(Locale locale, int strength)
                throws NullPointerException { … }
        public QueryCollator(Locale locale, int strength, int decomposition)
                throws NullPointerException { … }

        // methods
        public Locale getLocale() { … }
        public boolean hasStrength() { … }
        public int getStrength() { … }
        public boolean hasDecomposition() { … }
        public int getDecomposition() { … }

        public Collator getCollator() { … }

        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
}
```

- The constructors.
  The constructors create a new `QueryCollator` object with the locale specified by the `locale` argument. If the `strength` argument is specified, its value must be one of the `PRIMARY`, `SECONDARY`, `TERTIARY`, or `IDENTICAL` constants defined in the `java.text.Collator` class. If the `decomposition` argument is specified, its value must be one of the

[22] The `java.text.Collator` class is itself not serializable, hence cannot be used by remote SLEE management clients.

`NO_DECOMPOSITION`, `CANONICAL_DECOMPOSITION`, or `FULL_DECOMPOSITION` constants also defined in the `Collator` class.
The constructors may throw the following exception:

  - `java.lang.NullPointerException`.
    If the `locale` argument is null, a `NullPointerException` is thrown.

- The `getLocale` method.
  This method returns the `locale` argument of the constructor.
- The `hasStrength` and `getStrength` methods.
  The `hasStrength` method returns a `boolean` value indicating whether a collation strength was specified in the constructor. If this method returns `true`, the `getStrength` method returns the `strength` argument of the constructor. The return value of the `getStrength` method is undefined if `hasStrength` returns `false`.
- The `hasDecomposition` and `getDecomposition` methods.
  The `hasDecomposition` method returns a `boolean` value indicating whether a decomposition mode was specified in the constructor. If this method returns `true`, the `getDecomposition` method returns the `decomposition` argument of the constructor. The return value of the `getDecomposition` method is undefined if `hasDecomposition` returns `false`.
- The `getCollator` method.
  This method returns a `java.text.Collator` object equal to the collator described by this `QueryCollator` object.
- The `equals` method.
  This method indicates whether the specified object (specified by the `obj` argument) is equal to this `QueryCollator` object. The specified object is equal to this `QueryCollator` object if the specified object is a `QueryCollator` with the same locale, collation strength, and decomposition mode as this `QueryCollator`.
- The `hashCode` method.
  The implementation of this method returns the hash code value of the collator's locale.
- The `toString` method.
  This method returns a string representation of the `QueryCollator` object.

10.24 Profile Tables referenced by a Service

Each Service may optionally reference the following Profile Table.

- An Address Profile Table.
  This table contains provisioned addresses that may cause new root SBB entities of the Service to be instantiated.

Note: The Resource Info Profile Table that was defined in SLEE 1.0 has been deprecated in SLEE 1.1. A new version of the Address Profile Table is also defined by SLEE 1.1 to take into account the new Profile model.

10.24.1 Address Profile Table

Changed in 1.1: The Address Profile Table has been updated to support the SLEE 1.1 Profile query model. Refer to the 1.0 specification for the definition of the SLEE 1.0 Address Profile Table.

A Service may optionally reference an Address Profile Table. Any Profile Table that meets the following requirements can be a Service's Address Profile Table:

- Profiles within an Address Profile Table must include an `address` attribute. The Java type of this attribute is `javax.slee.Address`. (This should not be confused with the `address` attribute in the `ProfileID` class). Hence, the Profile CMP interface of the Profile Specification of the Profile Table must include the following methods:

```
public javax.slee.Address getAddress();
public void setAddress(javax.slee.Address address);
```

- The `address` attribute must be unique. This is specified using the Profile Specification's deployment descriptor (see Section 10.21). It is recommended that the `address` attribute has appropriate index-hints defined. As Address Profiles may be used for SLEE event routing it is strongly recommended that the index-hints specify an `equals` operator.

Typically, each Address Profile in a Service's Address Profile Table represents a subscriber of the Service.

See how the SLEE uses the Address Profile Table to establish event filters in Section 8.7.1 and to determine whether new root SBB entities of the Service should be instantiated to handle an event in Section 8.6.

10.24.1.1 Standard Address Profile Specification

Changed in 1.1: The standard Address Profile Specification has been updated to support the SLEE 1.1 Profile query model. Refer to the 1.0 specification for the standard definition of the SLEE 1.0 Address Profile.

The SLEE 1.1 specification defines a standard Address Profile Specification. The standard Address Profile Specification has the required single `addresses` attribute.

The Profile CMP interface of the standard Address Profile Specification is as follows:

```
package javax.slee.profile;

import javax.slee.Address;

public interface AddressProfile11CMP {
       public Address getAddress();
       public void setAddresses(Address address);
}
```

The Profile local interface of the standard Address Profile Specification is as follows:

```
package javax.slee.profile;

public interface AddressProfileLocal extends ProfileLocalObject, AddressProfile11CMP {
}
```

The Profile management interface of the standard Address Profile Specification is as follows:

```
package javax.slee.profile;

public interface AddressProfileManagement extends AddressProfile11CMP {
}
```

The `profile-spec` element that describes this Profile Specification is as follows

```
<profile-spec>
       …
       <profile-spec-name>
              AddressProfileSpec
       </profile-spec-name>
       <profile-spec-vendor>
              javax.slee
       </profile-spec-vendor>
       <profile-spec-version>
              1.1
       </profile-spec-version>
       <profile-classes>
              <profile-cmp-interface >
                     <description> … </description>
                     <profile-cmp-interface-name>
                            javax.slee.profile.AddressProfile11CMP
```

```
                        </profile-cmp-interface-name>
                        <cmp-field unique="True">
                                <cmp-field-name>address</cmp-field-name>
                                <index-hint query-operator="equals"/>
                        </cmp-field>
                </profile-cmp-interface>
                <profile-local-interface>
                        <description> … </description>
                        <profile-local-interface-name>
                                javax.slee.profile.AddressProfileLocal
                        </profile-local-interface-name>
                </profile-local-interface>
                <profile-management-interface>
                        <description> … </description>
                        <profile-management-interface-name>
                                javax.slee.profile.AddressProfileManagement
                        </profile-management-interface-name>
                </profile-management-interface>
                <profile-abstract-class>
                        <description> … </description>
                        <profile-abstract-class-name>
                                ...
                        </profile-abstract-class-name>
                </profile-abstract-class>
        </profile-classes>
</profile-spec>
```

10.24.1.2 A custom Address Profile Specification example

The following example illustrates a custom (i.e. non-standard) Address Profile Specification. This example extends the standard Address Profile Specification, however extension is not a requirement when defining a custom Address Profile Specification. The Profile CMP interface of this custom Address Profile Specification is as follows:

```
package com.foobar;

import javax.slee.Address;
import javax.slee.profile.AddressProfile11CMP;
import javax.slee.profile.ProfileID;

public interface BarAddressProfileCMP extends AddressProfile11CMP {
        public String getFooData();
        public void setFooData(String s);
        // pointer into a QOSProfile in another Profile Table
        public ProfileID getQOSProfile();
        public void setQOSProfile(ProfileID p);
}
```

The Profile Local interface of this Profile Specification is as follows:

```
package com.foobar;

import javax.slee.profile.ProfileLocalObject

public interface BarAddressProfileLocal extends ProfileLocalObject, BarAddressProfileCMP
{
}
```

The Profile Management interface of this Profile Specification is as follows:

```
package com.foobar;

public interface BarAddressProfileManagement extends BarAddressProfileCMP {
}
```

The Profile abstract class of this Profile Specification is as follows:

```
package com.foobar;

import javax.slee.profile.Profile;
```

```
import javax.slee.profile.ProfileContext;
import javax.slee.profile.ProfileLocalObject;
import javax.slee.profile.ProfileTable;
import javax.slee.profile.ProfileVerificationException;

public abstract class BarAddressProfileImpl
        implements Profile, BarAddressProfileCMP, BarAddressProfileManagement
{
        public void setProfileContext(ProfileContext context) {
                this.context = context;
                …
        }
        public void unsetProfileContext() { … }

        public void profileInitialize() { … }
        public void profilePostCreate() { … }

        public void profileActivate() { … }
        public void profileDeactivate() { … }
        public void profileLoad() { … }
        public void profileStore() { … }
        public void profileRemove() { … }

        public void profileVerify() throws ProfileVerificationException {
                …
                /* check QOS Profile is valid */
                ProfileID qosProfileID = getQOSProfile();
                ProfileTable qosProfileTable =
                        context.getProfileTable(qosProfileID.getProfileTableName());
                ProfileLocalObject qosProfile =
                        qosProfileTable.find(qosProfileID.getProfileName());
                if (qosProfile != null) …

                /* check addresses are valid */
                …
        }

        private ProfileContext context;
}
```

The `profile-spec` element that describes this Profile Specification is as follows:

```
<profile-spec>
        …
        <profile-spec-name>
                BarAddressProfileSpec
        </profile-spec-name>
        <profile-spec-vendor>
                com.foobar
        </profile-spec-vendor>
        <profile-spec-version>
                1.0
        </profile-spec-version>
        <profile-classes>
                <profile-cmp-interface >
                        <description> … </description>
                        <profile-cmp-interface-name>
                                com.foobar.BarAddressProfileCMP
                        </profile-cmp-interface-name>
                        <cmp-field unique="True">
                                <cmp-field-name>address</cmp-field-name>
                                <index-hint query-operator="equals"/>
                        </cmp-field>
                </profile-cmp-interface>
                <profile-local-interface>
                        <description> … </description>
                        <profile-local-interface-name>
                                com.foobar.BarAddressProfileLocal
                        </profile-local-interface-name>
                </profile-local-interface>
```

```
                <profile-management-interface>
                        <description> … </description>
                        <profile-management-interface-name>
                                com.foobar.BarAddressProfileManagement
                        </profile-management-interface-name>
                </profile-management-interface>
                <profile-abstract-class>
                        <description> … </description>
                        <profile-abstract-class-name>
                                com.foobar.BarAddressProfile
                        </profile-abstract-class-name>
                </profile-abstract-class>
        </profile-classes>
</profile-spec>
```

10.25 Deployment processing

At Profile Specification deployment time, the SLEE implements the following classes for the Profile Specification:

- The SLEE implements a class that implements the Profile CMP interface. This class is known as the Profile CMP implementation class.
    - The get accessor methods of this class are initialized to the most recent committed state and are part of a transactions isolated state. If SLEE components have a read-write view of the Profiles created from the Profile Specification (see Section 10.5.1.1) then the set accessor methods must allow the Profile to be updated within the context of the enclosing transaction. Otherwise if SLEE components have a read-only view of the Profiles, the set accessor methods must throw a `java.lang.UnsupportedOperationException`.
- The SLEE implements a class that extends the Profile abstract class if one is defined. Otherwise the SLEE implements a class that implements the Profile CMP interface and the `javax.slee.profile.Profile` interface. This class is known as the Profile concrete class. A Profile object is an instance of this class.
    - If a Profile abstract class has not been defined, the SLEE provides a default implementation of the methods defined in the `Profile` interface. If a Profile abstract class has been defined the SLEE expects that the Profile Specification Developer has implemented all the methods of the `Profile` interface in the Profile abstract class.
    - The SLEE must implement the set accessor methods inherited from the Profile CMP interface to return by throwing a `javax.slee.profile.ReadOnlyProfileException` if the set accessor methods are invoked on a Profile object from a Profile MBean object that is in a read-only state (see Section 10.28) or from a Profile local object when SLEE component has a read-only view of the Profile.
    - If the Profile abstract class defines any get Usage Parameters interface methods, the SLEE must implement these methods.
- The SLEE implements a class that implements the Profile local interface if one is defined. This class is known as the Profile local implementation class and is for the SLEE Component view of the Profile Specification. A Profile local object is an instance of this class.
    - The SLEE must implement all the methods defined in the `javax.slee.profile.ProfileLocalObject` interface in this class.
    - The SLEE must implement all the CMP accessor methods and/or business methods defined in the Profile local interface. Generally these methods delegate to the methods with the same signature in the Profile abstract class.

- The SLEE implements a class that implements the Profile Table interface is one is defined. This class is known as the Profile Table implementation class and is for the SLEE Component view of the Profile Specification. A Profile Table object is an instance of this class.
    - The SLEE must implement all the methods defined in the `javax.slee.profile.ProfileTable` interface in this class.
    - If a Profile Table interface that extends the `ProfileTable` interface has been defined, the SLEE must implement any static query methods defined by the Profile Specification in this class.
- The SLEE implements a Profile MBean class (see Section 10.26).

10.26 Profile MBean

The SLEE implements a Profile MBean class for each Profile Specification. A Profile MBean object allows an Administrator to access a Profile defined by the Profile Specification. More precisely, the Administrator invokes the Profile MBean object and the Profile MBean object invokes the Profile object that caches the persistent state of a Profile.

The Profile MBean class may implement any MBean type supported by the JMX 1.2.1 specification (e.g. standard, dynamic, model, etc), but must present an MBean interface to the MBean Server that follows the rules discussed in this section. This MBean interface is called the Profile MBean interface.

It is the responsibility of the Profile Specification Developer to ensure there are no method naming conflicts that may arise during the Profile MBean construction. If the SLEE encounters errors caused by duplicate method names, the SLEE should reject the Profile Specification deployment attempt.

10.26.1 Requirements for the Profile MBean interface

The following are the requirements for the Profile MBean interface:

- The Profile MBean interface must include in its definition all the methods defined in the interface `javax.slee.profile.ProfileMBean`, either by directly or indirectly inheriting from `ProfileMBean` interface or by including the attributes and operations defined in the `ProfileMBean` interface in its dynamic interface.
- If the Profile Specification defines a Profile Management interface, each method defined in the Profile Management interface must be present with the same name, argument, and return types in the Profile MBean interface. This method may be a CMP field accessor method or a management method.
- If the Profile Specification does not define a Profile Management interface, each method defined in the Profile CMP interface must be present with the same name, argument, and return types in the Profile MBean interface.
- The `throws` clause of each CMP field get accessor method that is present in the Profile MBean interface may include `javax.slee.management.ManagementException` if necessary.
- The `throws` clause of each CMP field set accessor method that is present in the Profile MBean interface must include `javax.slee.InvalidStateException`, and may include `javax.slee.management.ManagementException` if necessary.
- The `throws` clause of each management method that is present in the Profile MBean interface must include `javax.slee.InvalidStateException` and `javax.slee.profile.ProfileImplementationException`, and may include `javax.slee.management.ManagementException` if necessary.

10.26.2 Requirements for the Profile MBean class

The following are the requirements for the Profile MBean class:

- The Profile MBean class must be implemented in accordance with the JMX 1.2.1 specification.
- The Profile MBean class can be implemented as any type of MBean supported by the JMX 1.2.1 specification but must present the Profile MBean interface that is constructed using the rules specified in Section 10.26.1 to the MBean Server.
- The Profile MBean class must expose get and set accessor methods that follow the design pattern for managed attributes as attributes and it may expose them as managed operations, i.e. they may be accessible via the `invoke` method of the MBean Server, however they must be accessible via the `getAttribute` (for get accessor methods) and `setAttribute` (for write accessor methods) methods of the MBean Server[23].
- The Profile MBean class must implement all the methods defined in the `javax.slee.profile.ProfileMBean` interface. Section 10.26.3 describes each of these methods.
- The Profile MBean class must support the Profile MBean object life cycle defined in Section 10.28. If invocation of a set accessor method or a management method results in `javax.slee.profile.ProfileReadOnlyException` being thrown by the Profile concrete class (for example, because a write operation is attempted on a Profile MBean object that is in the read-only state), the SLEE translates the exception to a `javax.slee.InvalidStateException` before throwing it to the invoking management client.
- The Profile MBean class must implement all the methods that are included in the Profile MBean interface. The method implementations typically delegate invocations to appropriate methods in the Profile concrete class.
- If the implementation of a management method provided by a Profile Specification Developer throws a checked exception, the exception must be wrapped in a `javax.slee.profile.ProfileImplementationException` and re-thrown to the invoking management client.
- If the implementation of a management method provided by a Profile Specification Developer throws a runtime exception, the Profile object is discarded, the transaction is marked for rollback, and the exception must be wrapped in a `javax.slee.profile.ProfileImplementationException` and re-thrown to the invoking management client.
- The Profile MBean class may use `javax.slee.management.ManagementException` to report errors to the management client when an attempt to access or update persistent CMP fields fails due to a system-level problem.

10.26.3 `ProfileMBean` interface

Changed in 1.1: Defined the object names of all MBeans.

All Profile MBean classes implement the `javax.slee.profile.ProfileMBean` interface. The `ProfileMBean` interface defines the methods that support a transactional life cycle for Profile MBean objects.

The `ProfileMBean` interface is as follows:

```
package javax.slee.profile;

import javax.slee.InvalidStateException;
import javax.slee.management.ManagementException;

public interface ProfileMBean {
       // Base JMX Object Name string of a ProfileMBean
```

[23] JAIN SLEE 1.0 used the JMX 1.0 specification. JAIN SLEE 1.1 uses the JMX 1.2.1 specification. Under JMX 1.0 a management client could invoke the accessor methods via the MBean Server's `invoke` method as well as via the `getAttribute` and `setAttribute` methods. Under JMX 1.2.1 the default behaviour is that managed attributes can only be accessed via the MBean Server's `getAttribute` and `setAttribute` methods.

```
        public static final String BASE_OBJECT_NAME = "javax.slee.profile:type=Profile";

        // methods
        public void editProfile()             throws  ManagementException;
        public void commitProfile()           throws  ManagementException,
                                                      ProfileVerificationException,
                                                      InvalidStateException;
        public void restoreProfile()          throws  ManagementException,
                                                      InvalidStateException;
        public void closeProfile()            throws  ManagementException,
                                                      InvalidStateException;
        public boolean isProfileWriteable()   throws  ManagementException;
        public boolean isProfileDirty()       throws  ManagementException;
}
```

The `BASE_OBJECT_NAME` constant specifies the base JMX object name for all Profile MBeans.

10.26.3.1 The `editProfile` method

The Administrator invokes the `editProfile` method to obtain read-write access to the Profile MBean object (if the Administrator currently has read-only access to the Profile MBean object).

- The implementation of this method should start a new transaction for the editing session, or perform the equivalent function.
- If a system-level failure is encountered during the execution of this method or a concurrent edit of the same Profile is attempted on a SLEE implementation that does not support concurrent edits (see Section 10.29.2), the implementation should throw a `javax.slee.management.ManagementException` to report the exceptional situation to the Administrator.
- If the Profile MBean object is already in the read-write state when this method is invoked, this method has no further effect and returns silently.

10.26.3.2 The `commitProfile` method

Once the Administrator has used the Profile MBean object to make changes to the Profile object that caches the persistent state of a Profile, the Administrator invokes the `commitProfile` method to attempt to commit the changes permanently.

- The implementation of this method must invoke the `profileVerify` method of the Profile object that caches the persistent state of the Profile, and only commit the changes if the `profileVerify` method returns without throwing an exception. If the `profileVerify` method throws a `javax.slee.profile.ProfileVerificationException`, the commit attempt should fail, the exception is forwarded back to the management client, and the Profile MBean object must remain in the read-write state.
- The implementation of this method must also verify that the constraints specified by the Profile Specification's deployment descriptor, such as the uniqueness constraints placed on indexed attributes. The SLEE verifies these constraints after it invokes the `profileVerify` method of the Profile object. If any constraint is violated, then this method throws a `javax.slee.profile.ProfileVerificationException`, the commit attempt fails, and the Profile MBean object must remain in the read-write state.
- If a commit succeeds, the Profile MBean object returns to the read-only state. If the Profile MBean was created in response to the `createProfile` method being invoked on the Profile Provisioning MBean and the `profile-events-enabled` attribute of the Profile Specification is set to "True" (see Section 10.9.6), the SLEE must fire a Profile Added Event (see Section 10.9.2). If the Profile was an existing profile that has been updated and the `profile-events-enabled` attribute of the Profile Specification is set to "True", the SLEE must fire a Profile Updated Event (see Section 10.9.3).

- If a commit fails due to a system-level failure, the implementation of this method should throw a `javax.slee.management.ManagementException` to report the exceptional situation to the management client. The Profile MBean object should continue to remain in the read-write state if possible.
- The execution of the `commitProfile` method (and `profileVerify` callback methods of the Profile object) must run in the same transaction context as that begun by the `editProfile` invocation that initiated the editing session or the same transaction context as which the Profile was created using the `createProfile` method of the Profile Provisioning MBean. The transaction should only be committed if the `commitProfile` method returns successfully, i.e. without throwing an exception.
- The `commitProfile` method must throw a `javax.slee.InvalidStateException` if the Profile MBean object is not in the read-write state.

10.26.3.3 The `restoreProfile` method

The Administrator invokes the `restoreProfile` method if the Administrator wishes to discard changes made to the Profile..

- The implementation of this method rolls back any changes that have been made to the Profile since the Profile MBean object entered the read-write state and moves the Profile MBean object to the read-only state. If the Profile MBean object was returned by the Profile Provisioning MBean object's `createProfile` method (see Section □), then no new Profile is created since the transaction will not commit, and the Profile MBean is implicitly closed by the SLEE to deregister it from the MBean Server.
- The execution of this method must begin in the same transaction context as that begun by the `createProfile` or `editProfile` invocation that initiated the editing session, but must roll back the transaction before returning.
- The `restoreProfile` method must throw a `javax.slee.InvalidStateException` if the Profile MBean object is not in the read-write state.

10.26.3.4 The `closeProfile` method

The Administrator invokes the `closeProfile` method when the Administrator no longer requires access to the Profile MBean object.

- The implementation of this method is free to deregister the Profile MBean object from the MBean Server.
- The `closeProfile` method must throw a `javax.slee.InvalidStateException` if the Profile MBean object is in the read-write state and has uncommitted changes, i.e. the `isProfileDirty` method returns `true`.

10.26.3.5 The `isProfileWriteable` method

The `isProfileWriteable` method returns `true` if the Profile MBean object is in the read-write state, or `false` if in the read-only state.

10.26.3.6 The `isProfileDirty` method

The `isProfileDirty` method returns `true` if the Profile MBean object is in the read-write state and a Profile CMP field has been modified during the current Profile editing session. For Profile MBeans representing Profiles newly created by the Profile Provisioning MBean's `createProfile` method that have yet to committed, this method always returns `true`.

This method returns `false` under any other situation.

10.27 Life cycle of a Profile

A Profile either exists or does not exist. If a Profile exists, its persistent state exists and is maintained persistently by the SLEE.

A Profile Specification must be deployed before Profile Tables of the Profile Specification can be created. A Profile cannot exist before the Profile Table that it belongs to is created.

When a Profile Table is created, the default Profile is created and initialized. The default Profile is removed when the Profile Table is removed. Figure 16 illustrates the life cycle of the default Profile of a Profile Table, and the callbacks that are made on a Profile object that caches the persistent state of the Profile on life cycle transitions. The callbacks are made within the same (new) transaction context. The transaction commits after the `profilePostCreate` callback has successfully returned. (Other lifecycle methods such as `profileStore` may also be invoked on the Profile object before the transaction commits in order to synchronize the persistent state of the Profile with any transient state stored in the Profile object.)

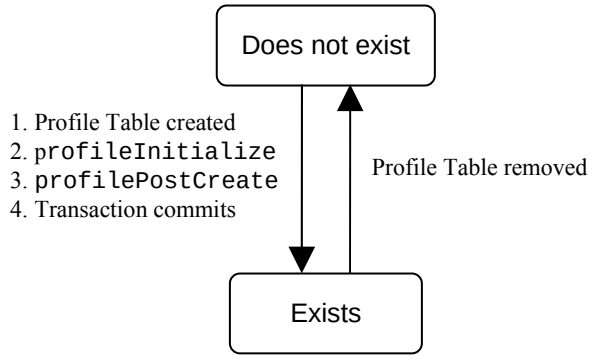

Figure 16 Life cycle of the default Profile
All specified methods are on the `Profile` interface

A named (i.e. user-defined) Profile is explicitly created and removed by an Administrator through the SLEE's Profile Provisioning MBean interface, or by a SLEE Component through the `ProfileTable` interface. A new Profile's persistent state is initialized with a copy of the persistent state of the default Profile and the management client or SLEE component that created the Profile may modify this initial state before the transaction that is creating the Profile commits. Figure 17 illustrates the life cycle of a named Profile.

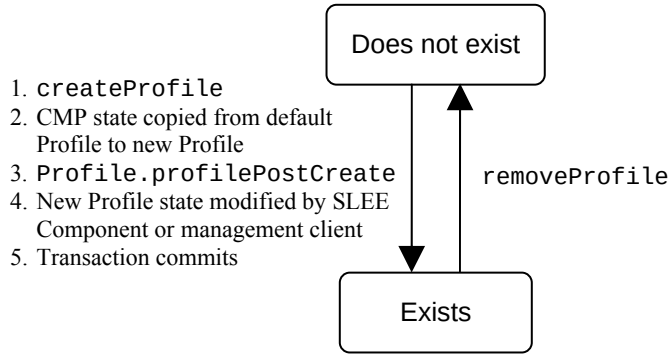

Figure 17 Life cycle of a named Profile
All specified methods unless otherwise stated are on the `ProfileProvisioningMBean` interface

10.28 Life cycle of Profile MBean objects

A Profile MBean object is an instance of a Profile MBean and it represents a Profile and allows the Profile to be accessed by an Administrator. It has access to the Profile object that caches the persistent state of the Profile and any other objects that are required by the SLEE to implement functionality of the Profile MBean object. A Profile MBean object must be registered with the SLEE's JMX MBean Server in order to be accessible to management clients.

The JAIN SLEE specification does not dictate when a Profile MBean object or a Profile object must be created or destroyed. However, an appropriate Profile MBean object must be registered with the MBean server when the JMX Object Name of a Profile is requested via the `ProfileProvisioningMBean` interface and may be deregistered when the `closeProfile` method is invoked on the Profile MBean object.

When a Profile is created, the Profile MBean object for the new Profile must portray a *read-write* view to management clients. Alternatively, when the `editProfile` method is invoked on a Profile MBean object, the Profile MBean object also shifts to the read-write state.

- If the `restoreProfile` method is invoked on the Profile MBean object, any changes that have been made to the Profile object that caches the persistent state of the Profile since the Profile MBean object entered the read-write state must be discarded.
- If instead the `commitProfile` method is invoked on the Profile MBean object, the `profileVerify` callback method is invoked on the Profile object that caches the persistent data of the Profile. If the verification callback returns without throwing an exception and the Profile MBean object passes additional SLEE implemented verification (such as checking for uniqueness of "unique" indexed attributes), the changes made to the Profile object are committed and the Profile MBean object moves to the read-only state. If an exception is thrown by the verification callback, the Profile MBean object remains in the read-write state if possible.
- The SLEE may invoke the `profileLoad` and `profileStore` callback methods at any time to synchronize the transient state held in the Profile object with the persistent state cached in CMP fields. If the Profile MBean object is in the read-write state, then the SLEE invokes the `profileLoad` and `profileStore` methods with the same transaction context as management methods on the Profile object.

Figure 18 illustrates the life cycle of a Profile MBean object.

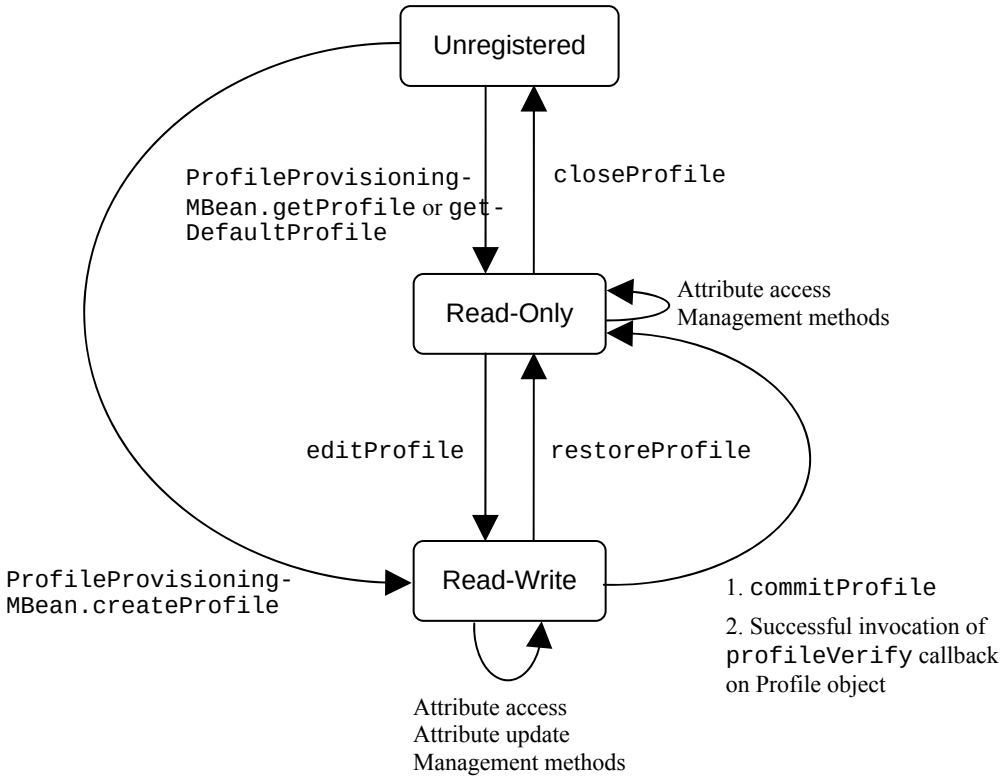

Figure 18 Life Cycle of a Profile MBean object
All specified methods unless otherwise stated are on the `ProfileMBean` interface

10.29 Profile MBean object management

Profile MBean objects do not always have to be registered with the SLEE's MBean Server. The Administrator can only access a Profile after it invokes the `createProfile`, `getProfile`, and `getDefaultProfile` methods of the `ProfileProvisioningMBean` interface (see Section 14.10).

Each of these methods returns a suitable JMX Object Name for the appropriate Profile. However, before returning the Object Name from one of these methods, the SLEE must ensure that the appropriate Profile exists and the Profile MBean object that represents the Profile is registered with the MBean server.

Once finished with the Profile, the Administrator can invoke the `closeProfile` method directly on the Profile MBean object that represents the Profile. This informs the SLEE that the Profile is no longer required by the Administrator, and the Profile MBean object may be deregistered from the MBean Server if no other valid references to the Profile MBean object's Object Name have been given to other Administrators.

After invoking a `closeProfile` method on a Profile MBean object, the Administrator must assume that the Object Name reference previously obtained is no longer valid.

10.29.1 Profile editing

When the Administrator invokes the `getProfile` and `getDefaultProfile` methods of a `ProfileProvisioningMBean` object to get access to a Profile (see Section 14.10), the Administrator initially has read-only access to the Profile. The Administrator invokes the `editProfile`, `commitProfile`, and `restoreProfile` methods on the Profile MBean object to manage the life cycle of the Profile MBean object.

When the Administrator creates a new Profile, the Administrator has read-write access to a Profile MBean object, equivalent to invoking `editProfile` on the Profile MBean object. The initial attribute values of the new Profile MBean object are obtained from the default Profile. The SLEE will try to add the new Profile to the Profile Table only after the Administrator invokes the `commitProfile` method on the Profile MBean object. This allows the Profile MBean object and the SLEE to verify the contents of new Profile before adding the Profile to the Profile Table.

10.29.2 Management client isolation

A SLEE implementation may allow two or more management client sessions to edit a Profile concurrently. If a Profile is edited simultaneously in two or more separate sessions, the SLEE must ensure that any changes made to the Profile in one session are isolated from (i.e. not visible to) the other sessions. Whether or not concurrent transactions against the Profile are able to commit depends on the concurrency control semantics used by the SLEE implementation, and the semantics of the concurrent access to the Profile.

Alternatively, a SLEE implementation may not allow two or more sessions to edit a Profile concurrently. If this occurs, the `editProfile` method of the `ProfileMBean` interface throws a `javax.slee.management.ManagementException`.

10.30 Component object view updates

Any changes made to a Profile by an Administrator must not be made visible to SLEE component objects (e.g. Resource Adaptor and SBB objects) until the Administrator commits these changes and the SLEE commits these changes successfully.

Chapter 11 Usage Parameters

Changed in 1.1: The domain of usage parameters has been expanded to allow more SLEE Components and SLEE internal subsystems to expose usage information. SLEE Components may read Usage Parameters. The name used to address each usage parameter has been changed to match its notification source.

A usage parameter is a parameter that can be updated by an object in the SLEE to provide usage information and is accessible by the Administrator through the management interface of the SLEE.

Logically, each usage parameter is addressed by a hierarchically scoped name. The three hierarchically scoped name components within the fully scoped name are as follows:

- Notification source name component.
  This name component is the top-most scoped name. It identifies the SLEE entity, such as an SBB, Profile Table, or resource adaptor entity, or SLEE internal subsystem that the usage parameter belongs to. (See Section 14.16 for more information on notification sources.)
- Usage parameter set name component.
  This name component is the second-level scoped name. It identifies a usage parameter set (see Section 11.3) accessible by the notification source. This name component is undefined when the usage parameter identified by the usage parameter name component is in the default usage parameter set for the notification source.
- Usage parameter name component.
  This name component is the lowest-level scoped name. It identifies a usage parameter in the usage parameter set of the notification source identified by the notification source name component.

11.1 Usage parameter types

There are two types of usage parameters.

- Counter-type usage parameters.
  A counter-type usage parameter can be incremented or decremented[24]. A SLEE Component may get the current (approximate) value of a counter-type usage parameter through the Usage Parameters interface. The Administrator can get and reset the value of a counter-type usage parameter through the SLEE's management interface.
- Sample-type usage parameters.
  A sample-type usage parameter accumulates sample data. A SLEE Component may submit samples to the usage parameter and get the the current (approximate) minimum, maximum, mean, and number of sample values added to a sample-type usage parameter through the Usage Parameters interface. The Adminstrator can get the minimum, maximum, mean, and number of the sample values added to a sample-type usage parameter and reset the state of the usage parameter through the SLEE's management interface.

11.2 Usage Parameters interfaces

SLEE Components declare their usage parameters via a Usage Parameters interface[25]. An SBB Developer may optionally declare a Usage Parameters interface for an SBB. A Profile Specification Developer may optionally declare a Usage Parameters interface for a Profile Specification. A Resource Adaptor Developer may optionally declare a Usage Parameters interface for a Resource Adaptor.

- The Usage Parameters interface must be defined in a named package, i.e. the class must have a `package` declaration. *(Added in 1.1)*

[24] Decrementing can be achieved by passing a negative value to an increment method.
[25] The SLEE specification does not define how SLEE internal subsystems declare their usage parameters, but does define how they are managed by the Administrator (see Section 14.9).

- The Usage Parameters interface must be declared as `public`.
- Each increment or sample method within the Usage Parameters interface declares a lowest level usage parameter name relevant to the SLEE Component.
    - The SLEE derives the usage parameter type associated with this usage parameter name from the method name of the declared method.
- Each get accessor method within the Usage Parameters interface provides access to the current approximate value or sample statistics for the lowest-level usage parameter name.
- A single usage parameter name can only be associated with a single usage parameter type. The SLEE must check and reject a Usage Parameters interface that declares both an increment method and a sample method for the same usage parameter name.
- A usage parameter name must be valid Java identifier and begins with a lowercase letter, as determined by `java.lang.Character.isLowerCase.`
- This interface is optional. If the SLEE Component does not declare any usage parameter names, a Usage Parameters interface does not have to be provided for the SLEE Component. *(Clarified in 1.1)*

11.2.1 Counter-type usage parameter increment method

A usage parameter increment method is defined in the Usage Parameters interface to declare the presence of and to permit updates to a counter-type usage parameter. The method name of the increment method is derived by adding an "increment" prefix to the usage parameter name. The increment method has the following method signature:

```
public abstract void increment<usage parameter name>(long value);
```

- The increment method must be declared as `public` and `abstract`.
- The first letter of the usage parameter name is uppercased in the definition of the increment method.
- The increment method does not have a `throws` clause.
- This method runs in an unspecified transaction context (see Section 9.7.1). Counter-type usage parameter updates do not require an active transaction. Counter-type usage parameter updates occur regardless of the outcome of any transaction active at the time of the update. If multiple threads update the same usage parameter at the same time these updates must be accumulated as if the updates were serial.
- The method throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

11.2.2 Counter-type usage parameter accessor method

Added in 1.1

A usage parameter accessor method is defined in the Usage Parameters interface to provide access to a counter-type usage parameter. The method name of the accessor method is derived by adding a "get" prefix to the usage parameter name. The accessor method has the following method signature:

```
public abstract long get<usage parameter name>();
```

- The accessor method must be declared as `public` and `abstract`.
- The return type of the method must be `long`.
- The first letter of the usage parameter name is uppercased in the definition of the accessor method.
- The method does not have a `throws` clause.

- This method runs in an unspecified transaction context (see Section 9.7.1). Counter-type usage parameter access does not require an active transaction.
- Since updates to a usage parameter occur non-transactionally, and may not be applied immediately due to concurrency serialization issues, a SLEE component can only assume the value returned from this method to be an approximation of the true value of the usage parameter at the point in time when the accessor method is invoked.
- The method throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

11.2.3 Sample-type usage parameter sample method

A usage parameter sample method is defined in the Usage Parameters interface to declare the presence of and to permit updates to a sample-type usage parameter. The method name of the sample method is derived by adding a "sample" prefix to the usage parameter name. The sample method has the following method signature:

```
public abstract void sample<usage parameter name>(long value);
```

- The sample method must be declared as `public` and `abstract`.
- The first letter of the usage parameter name is uppercased in the definition of the sample method.
- The sample method does not have a `throws` clause.
- This method runs in an unspecified transaction context (see Section 9.7.1). Sample-type usage parameter updates do not require an active transaction. Sample-type usage parameter updates occur regardless of the outcome of any transaction active at the time of the update. If multiple threads update the same usage parameter at the same time these updates must be accumulated as if the updates were serial.
- The method throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

11.2.4 Sample-type usage parameter accessor method

Added in 1.1

A sample-type usage parameter accessor method is defined in the Usage Parameters interface to provide access to a sample-type usage parameter. The method name of the accessor method is derived by adding a "get" prefix to the usage parameter name. The accessor method has the following method signature:

```
public abstract SampleStatistics get<usage parameter name>();
```

- The accessor method must be declared as `public` and `abstract`.
- The return type of the method must be `javax.slee.usage.SampleStatistics`.
- The first letter of the usage parameter name is uppercased in the definition of the accessor method.
- The method does not have a `throws` clause.
- This method runs in an unspecified transaction context (see Section 9.7.1). Sample-type usage parameter access does not require an active transaction.
- Since updates to a usage parameter occur non-transactionally, and may not be applied immediately due to concurrency serialization issues, a SLEE component can only assume the statistics returned from this method to be an approximation of the true values for the usage parameter at the point in time when the accessor method is invoked.
- The method throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

11.2.5 SBB Usage Parameters interface deployment descriptor

If an SBB declares a Usage Parameters interface, the deployment descriptor of the SBB must identify the Usage Parameters interface. The `sbb-usage-parameters-interface` element of the SBB deployment descriptor identifies this interface. It contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- An `sbb-usage-parameters-interface-name` element.
  This element identifies the class name of the Usage Parameters interface.
- Zero or more `usage-parameter` elements. *(Added in 1.1)*
  These elements allow specified properties of each usage parameter to be declared at deployment time. It declares the following attributes:
  - The `name` attribute.
    This attribute specifies the name of the usage parameter.
  - The `notifications-enabled` attribute.
    This boolean attribute specifies whether the generation of usage notifications will initially be enabled or disabled for the usage parameter. By default, the generation of usage notifications is disabled for all usage parameters. The generation of usage notifications can be enabled and disabled after deployment by invoking methods on the Usage Notification Manager MBean generated by the SLEE for the Usage Parameters interface.
    Note: If an SBB that defines a Usage Parameters interface is installed using a SLEE 1.0 deployment descriptor, the SLEE must assume the behavior of the SLEE 1.0 specification and enable *the generation of usage notifications for the usage parameters defined by the SBB in the Usage Parameters interface by default.*

11.2.6 Profile Specification Usage Parameters interface deployment descriptor

Added in 1.1.

If a Profile Specification declares a Usage Parameters interface, the deployment descriptor of the Profile Specification must identify the Usage Parameters interface. The `profile-usage-parameters-interface` element of the Profile Specification deployment descriptor identifies this interface. It contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- A `profile-usage-parameters-interface-name` element.
  This element identifies the class name of the Usage Parameters interface.
- Zero or more `usage-parameter` elements.
  These elements allow specified properties of each usage parameter to be declared at deployment time. It declares the following attributes:
  - The `name` attribute.
    This attribute specifies the name of the usage parameter.
  - The `notifications-enabled` attribute.
    This boolean attribute specifies whether the generation of usage notifications will initially be enabled or disabled for the usage parameter. By default, the generation of usage notifications is disabled for all usage parameters. The generation of usage notifications can be enabled and disabled after deployment by invoking methods on the Usage Notification Manager MBean generated by the SLEE for the Usage Parameters interface.

11.2.7 Resource Adaptor Usage Parameters interface deployment descriptor

Added in 1.1.

If a Resource Adaptor declares a Usage Parameters interface, the deployment descriptor of the Resource Adaptor must identify the Usage Parameters interface. The `resource-adaptor-usage-parameters-interface` element of the Resource Adaptor deployment descriptor identifies this interface. It contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- A `resource-adaptor-usage-parameters-interface-name` element.
  This element identifies the class name of the Usage Parameters interface.
- Zero or more `usage-parameter` elements.
  These elements allow specified properties of each usage parameter to be declared at deployment time. It declares the following attributes:
  - The `name` attribute.
    This attribute specifies the name of the usage parameter.
  - The `notifications-enabled` attribute.
    This boolean attribute specifies whether the generation of usage notifications will initially be enabled or disabled for the usage parameter. By default, the generation of usage notifications is disabled for all usage parameters. The generation of usage notifications can be enabled and disabled after deployment by invoking methods on the Usage Notification Manager MBean generated by the SLEE for the Usage Parameters interface.

11.2.8 Usage Parameters interface example

In the following example, the `FooSbbUsageParameters` interface declares two usage parameter names for counter-type usage parameters and two usage parameter names for sample-type usage parameters relevant the the Foo SBB.

```
package com.foobar;
…
public interface FooSbbUsageParameters {
        // counter-type usage parameter names
        public void incrementFirstCount(long value);
        public void incrementSecondCount(long value);

        // sample-type usage parameter names
        public void sampleTimeBetweenNewConnections(long value);
        public void sampleTimeBetweenErrors(long value);
}
```

The usage parameter names of the counter-type usage parameters are “firstCount” and “secondCount”. The usage parameter names of the sample-type usage parameters are “timeBetweenNewConnections” and “timeBetweenErrors”. Usage notifications will be initially disabled for the counter-type usage parameters, but enabled for the sample-type usage parameter. *(Added in 1.1)*

```
<sbb>
        …
        <sbb-classes>
                …
                <sbb-usage-parameters-interface>
                        <description> … </description>
                        <sbb-usage-parameters-interface-name>
                                com.foobar.FooSbbUsageParameter
                        </sbb-usage-parameters-interface-name>
                        <usage-parameter name="firstCount" notifications-enabled="False"/>
                        <usage-parameter name="secondCount" notifications-enabled="False"/>
                        <usage-parameter name="timeBetweenNewConnections"
```

```
                                    notifications-enabled="True"/>
                        <usage-parameter name="timeBetweenErrors"
                                    notifications-enabled="True"/>
                </sbb-usage-parameters-interface>
                …
        </sbb-classes>
        …
</sbb>
```

11.3 Usage parameter sets

A generator of usage information may access multiple usage parameters with the same lowest-level usage parameter name component by using multiple usage parameter sets. A usage parameter set is a set that contains a usage parameter for each usage parameter name declared in the Usage Parameters interface of the corresponding SLEE Component. Each method of the Usage Parameters interface declares the usage parameter name and type of a single usage parameter in this set.

Each usage parameter set has a name. Since a usage parameter set name component is scoped within a notification source name component, the usage parameter set namespace of a notification source is independent of the usage parameter set namespaces of a different notification source. Different SBBs within the same service are considered different notification sources, as is the same SBB used in multiple different Services.

For example, if Foo SBB is a member of the Bar Service and the Foo SBB declares the example Usage Parameters interface in Section 11.2.8, then each Foo SBB usage parameter set will include two counter-type usage parameters named “firstCount” and “secondCount” and two sample-type usage parameters named “timeBetweenNewConnections” and “timeBetweenErrors”. If the Foo SBB within the Bar Service has a usage parameter set named “GoldSet”, then all the Foo SBB objects within the Bar Service use this name to lookup this usage parameter set. SBB objects belonging to other SBBs within the Bar Service cannot get access to this usage parameter set. SBB objects belonging to other Services also cannot get access to this usage parameter set.

11.3.1 Default usage parameter set

By default, if a SLEE Component declares a Usage Parameters interface, the SLEE automatically maintains a default usage parameter set for each unique notification source derived from the SLEE Component. For example, if an SBB defines a Usage Parameters interface, the SLEE maintains a default usage parameter set for each Service the SBB is used in. If a Profile Specification defines a Usage Parameters interface, the SLEE maintains a default usage parameter set for each Profile Table created from the Profile Specification. If a Resource Adaptor defines a Usage Parameters interface, the SLEE maintains a default usage parameter set for each resource adaptor entity created from the Resource Adaptor.

The SLEE specification does not specify the name of the default usage parameter set. The only requirement is that that the name of the default usage parameter set is not a valid name that can be provided by an Administrator.

11.3.2 Named usage parameter sets

In addition to the default usage parameter set created by the SLEE for a notification source, the Administrator can create additional usage parameter sets for the same notification source. These Administrator created usage parameter sets must have a name.

- The name of a named usage parameter set may only contain letter or digit characters as defined by `java.lang.Character.isLetterOrDigit`. Additionally, any other character in the Unicode range 0x0020-0x007e may be included in a name.
- The name of a named usage parameter set cannot be zero-length.

11.4 Getting access to a usage parameter set

A usage parameter set is manipulated by a notification source using an object implemented by the SLEE which implements the Usage Parameters interface defined by the SLEE Component the notification source

was derived from. Each of these Usage Parameters objects represent a single usage parameter set and each of its increment or sample method manipulate a single counter-type or sample-type usage parameter in the usage parameter set.

11.4.1 SBB access to a usage parameter set

SBBs get access to a usage parameter set using get SBB Usage Parameter methods declared in the SBB abstract class. There are two forms of the get SBB Usage Parameter methods. The first form returns the default usage parameter set and the second form returns a named usage parameter set. The method signatures of these methods are as follow:

```
        public abstract <SBB Usage Parameters interface>
                                getDefaultSbbUsageParameterSet();
        public abstract <SBB Usage Parameters interface>
                                 getSbbUsageParameterSet(String name)
                throws UnrecognizedUsageParameterSetNameException;
```

- The methods must be declared as `public` and `abstract`. The SLEE provides the concrete implementation of the get SBB Usage Parameter methods when the SBB is deployed.
- Both get SBB Usage Parameter methods do not have to be declared if the SBB only requires use of one of the methods.
- The fully scoped name used to locate the usage parameter set consists of a notification source name component that encapsulates the SBB Component identifier of the SBB and the Service Component identifier of the Service the SBB is executing on behalf of and the usage parameter set name specified by the `name` argument (if the second method variant is used).
- The `getSbbUsageParameterSet` method throws a `java.lang.NullPointerException` if the `name` argument is `null`. This exception is not declared in the `throws` clause.
- The `getSbbUsageParameterSet` method throws a `javax.slee.usage.UnrecognizedUsageParameterSetNameException` if the `name` argument does not identify a named usage parameter set created by the Administrator for the SBB. *(Clarified in 1.1)*
- The return type must be the Usage Parameters interface of the SBB, or a base interface of the Usage Parameters interface of the SBB. The SBB uses the returned Usage Parameters object to update the usage parameters within the usage parameter set represented by the Usage Parameters object.
- These methods run in an unspecified transaction context (see Section 9.7.1). Getting access to a Usage Parameters object does not require an active transaction.
- These methods throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure. This exception is not declared in the `throws` clause.

11.4.1.1 Get SBB Usage Parameter method example

The get SBB Usage Parameter methods for the `FooSbbUsageParameter` interface may be as follow:

```
package com.foobar;
…
import javax.slee.Sbb;
import javax.slee.usage.UnrecognizedUsageParameterSetNameException;
…
public abstract class FooSbb implements Sbb … {
        …
        public abstract FooSbbUsageParameter getDefaultSbbUsageParameterSet();
        public abstract FooSbbUsageParameter getSbbUsageParameterSet(String name)
                throws UnrecognizedUsageParameterSetNameException;
        …
}
```

11.4.2 Profile access to a usage parameter set

Added in 1.1.

Profiles get access to a usage parameter set using get Usage Parameter methods declared in the Profile abstract class. There are two forms of the get Usage Parameter methods. The first form returns the default usage parameter set and the second form returns a named usage parameter set. The method signatures of these methods are as follow:

```
        public abstract <Profile Specification Usage Parameters interface>
                               getDefaultUsageParameterSet();
        public abstract <Profile Specification Usage Parameters interface>
                                getUsageParameterSet(String name)
                throws UnrecognizedUsageParameterSetNameException;
```

- The methods must be declared as `public` and `abstract`. The SLEE provides the concrete implementation of the get Usage Parameter methods when the Profile Specification is deployed.
- Both get Usage Parameter methods do not have to be declared if the Profile Specification only requires use of one of the methods.
- The fully scoped name used to locate the usage parameter set consists of a notification source name component that encapsulates the name of the Profile Table the Profile object is executing on behalf of and the usage parameter set name specified by the `name` argument (if the second method variant is used).
- The `getUsageParameterSet` method throws a `java.lang.NullPointerException` if the `name` argument is `null`. This exception is not declared in the `throws` clause.
- The `getUsageParameterSet` method throws a `javax.slee.usage.UnrecognizedUsageParameterSetNameException` if the `name` argument does not identify a named usage parameter set created by the Administrator for the Profile Table.
- The return type must be the Usage Parameters interface of the Profile Specification, or a base interface of the Usage Parameters interface of the Profile Specification. The Profile object uses the returned Usage Parameters object to update the usage parameters within the usage parameter set represented by the Usage Parameters object.
- These methods run in an unspecified transaction context (see Section 9.7.1). Getting access to a Usage Parameters object does not require an active transaction.
- These methods throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure. This exception is not declared in the `throws` clause.

11.4.2.1 Profile Get Usage Parameter method example

The Profile Get Usage Parameter methods for an example `FooProfileUsageParameter` interface may be as follow:

```
package com.foobar;
…
import javax.slee.profile.Profile;
import javax.slee.usage.UnrecognizedUsageParameterSetNameException;
…
public abstract class FooProfileImpl implements FooProfileCMP, Profile … {
        …
        public abstract FooProfileUsageParameter getDefaultUsageParameterSet();
        public abstract FooProfileUsageParameter getUsageParameterSet(String name)
                throws UnrecognizedUsageParameterSetNameException;
        …
}
```

11.4.3 Resource adaptor entity access to a usage parameter set

Added in 1.1.

Resource adaptor entities get access to a usage parameter set using the get Usage Parameter methods declared on the SLEE-defined `ResourceAdaptorContext` interface. There are two forms of the get Usage Parameter methods. The first form returns the default usage parameter set and the second form returns a named usage parameter set. The method signatures of these methods are as follow:

```
package javax.slee.resource;

import javax.slee.SLEEException;
import javax.slee.usage.NoUsageParametersInterfaceDefinedException;

public interface ResourceAdaptorContext {
        …
        public Object getDefaultUsageParameterSet()
                throws  NoUsageParametersInterfaceDefinedException,
                        SLEEException;
        public Object getUsageParameterSet(String name)
                throws  NullPointerException,
                        NoUsageParametersInterfaceDefinedException,
                        UnrecognizedUsageParameterSetNameException,
                        SLEEException;
}
```

- The fully scoped name used to locate the usage parameter set consists of a notification source name component that encapsulates the name of the resource adaptor entity the `ResourceAdaptorContext` object was created for and the usage parameter set name specified by the `name` argument (if the second method variant is used).
- The methods throw a `NoUsageParametersInterfaceDefinedException` if no Usage Parameters interface was declared in the deployment descriptor of the Resource Adaptor the resource adaptor entity was created from.
- The `getUsageParameterSet` method throws a `java.lang.NullPointerException` if the `name` argument is `null`.
- The `getUsageParameterSet` method throws a `javax.slee.usage.UnrecognizedUsageParameterSetNameException` if the `name` argument does not identify a named usage parameter set created by the Administrator for the resource adaptor entity.
- The methods return a `java.lang.Object` that may be Java typecast to the Usage Parameters interface declared by the Resource Adaptor. The resource adaptor entity uses the returned Usage Parameters object to update the usage parameters within the usage parameter set represented by the Usage Parameters object.
- These methods are non-transactional and run in an unspecified transaction context (see Section 9.7.1). Getting access to a Usage Parameters object does not require an active transaction.
- These methods throws a `javax.slee.SLEEException` if the requested operation cannot be performed due to a system-level failure.

Chapter 12 Runtime Environment

This chapter defines the application programming interfaces (APIs) that a compliant SLEE must make available to the instances of SLEE component classes at runtime. These APIs can be used by portable SLEE components because the APIs are guaranteed to be available in all SLEEs.

This chapter also defines the restrictions that the SLEE vendor can impose on the functionality that it provides to the SLEE components. These restrictions are necessary to enforce security and to allow the SLEE to properly manage the runtime environment.

12.1 APIs provided by SLEE

This section defines the minimal runtime environment that a SLEE makes available to the instances of SLEE component classes. The requirements described here are considered to be the minimal requirements; a SLEE implementation may choose to provide additional functionality that is not required by the SLEE specification.

A SLEE must make the following APIs available to the instances of SLEE component classes at runtime:

- Java 2 Platform, Standard Edition, v1.4 (J2SE) APIs (or later compatible versions) *(Changed in 1.1: Updated from J2SE v1.3)*
- SLEE 1.1 APIs
- JNDI 1.2 Standard Extension
- JMX 1.2.1 APIs
- JAXP 1.0
- JDBC 2.0 Standard Extension (support for row sets only)

12.1.1 Java 2 APIs

Changed in 1.1: Updated from J2SE v1.3

The SLEE must provide the full set of Java 2 Platform, Standard Edition, v1.4 (J2SE) APIs (or a later compatible version). The SLEE is not allowed to subset the Java 2 platform APIs. The SLEE is allowed to make certain Java 2 platform functionality unavailable to the instances of SLEE component classes by using the Java 2 platform security policy mechanism. The primary reason for the SLEE to make certain functions unavailable to these instances is to protect the security and integrity of the SLEE environment, and to prevent these instances from interfering with the SLEE's functions. The SLEE specification version 1.1 does not use any language features or APIs defined post J2SE v1.4. A SLEE implementation may use another version of the Java 2 Platform, Standard Edition provided that it provides support for the APIs defined in J2SE v1.4.

12.1.1.1 SLEE Component security permissions

The following table defines the Java 2 platform security permissions that the SLEE must be able to grant to the instances of SLEE component classes at runtime. The term "grant" means that the SLEE must be able to grant the permission, the term "deny" means that the SLEE should deny the permission.

Permission name	***SLEE policy***
java.security.AllPermission	Deny
java.awt.AWTPermission	Deny
java.io.FilePermission	Deny
java.net.NetPermission	Deny
java.util.PropertyPermission	Grant "read", "*" Deny all other
java.lang.reflect.ReflectPermission	Deny

java.lang.RuntimePermission	Deny
java.lang.SecurityPermission	Deny
java.io.SerializablePermission	Deny
java.net.SocketPermission	Deny

Changed in 1.1: The following security extensions have been added in 1.1.

SBB, Profile Specification, Resource Adaptor, and library components can be granted additional security permissions over and above the default set of security permissions granted by the SLEE using the `security-permissions` element in their respective deployment descriptor. Each `security-permissions` element contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- A `security-permission-spec` element.
  This element identifies the security permission policies used by component jar file classes. The syntax of this element is defined at the following URL:
  `http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html#FileSyntax`.
  Note: If the `codeBase` argument is not specified for a `grant` entry in the provided security permission specification, the SLEE assumes the code base to be the component jar file, and the security permissions specified in the `grant` entry are granted to all classes loaded from the component jar file, i.e. they apply to all SLEE Components defined in the component jar file. The security permissions are not granted to classes loaded from any other dependent component jar required by the components defined in the deployment descriptor. If the `codeBase` argument is specified for a `grant` entry, the specified path is taken to be relative to the root directory of the component jar within the deployable unit jar, but its use is otherwise undefined by the SLEE specification.

An example of a Resource Adaptor component jar deployment descriptor that defines additional security permissions is as follows:

```
<resource-adaptor-jar>
       <resource-adaptor>
              <description> … </description>
              <resource-adaptor-name> Foo JCC </resource-adaptor-name>
              <resource-adaptor-vendor> com.foo </resource-adaptor-vendor>
              <resource-adaptor-version> 10/10/20 </resource-adaptor-version>
              …
       </resource-adaptor>

       <security-permissions>
              <description>
                     Allow the resource adaptor to modify thread groups and connect to
                     remotehost on port 1234
              </description>
              <security-permission-spec>
                  grant {
                      permission java.lang.RuntimePermission "modifyThreadGroup";
                      permission java.net.SocketPermission "remotehost:1234", "connect";
                  };
              </security-permission-spec>
       </security-permissions>
</resource-adaptor-jar>
```

The following security requirements apply to methods invoked on classes loaded from an SBB component jar:

- Event-handler and initial event selector methods run with the default set of security permissions granted by the SLEE, plus any additional security permissions specified in the SBB's component jar deployment descriptor.

- The `isolate-security-permissions` attribute of the `sbb-local-interface` element in the SBB's deployment descriptor controls whether or not security permissions of other protection domains in the call stack are propagated to the SBB when a business method on the SBB Local interface is invoked. If the value of this attribute is "False", then the method in the SBB abstract class invoked as a result of a business method invoked on the SBB Local interface runs with an access control context that includes the protection domain(s) of the SBB as well as the protection domains of any other classes in the call stack as prescribed by the Java security model, such as the SBB that invoked the SBB Local interface method. If the value of the `isolate-security-permissions` attribute is "True", the SLEE automatically wraps the method invoked on the SBB abstract class in response to the SBB Local interface method invocation in an `AccessController.doPrivileged` block in order to isolate the security permissions of the invoked SBB, i.e. the security permissions of other protection domains in the call stack do not affect the invoked SBB.
- All methods defined in the `javax.slee.Sbb` interface can be invoked on an SBB object from an unpredictable call path. If any of these methods need to execute privileged code requiring security permissions over and above the standard set of permissions granted by the SLEE, the additional security permissions must be declared in the SBB component jar's deployment descriptor and the relevant methods must wrap the privileged code in an `AccessController.doPrivileged` block to ensure that potentially more restrictive security permissions of other protection domains in the call stack do not prohibit the privileged code from being executed.

The following security requirements apply to methods invoked on classes loaded from a Profile Specification component jar:

- All management methods invoked on the Profile Management interface run with the default set of security permissions granted by the SLEE, plus any additional security permissions specified in the Profile Specification's component jar deployment descriptor.
- The `isolate-security-permissions` attribute of the `profile-local-interface` element in the Profile Specification's deployment descriptor controls whether or not security permissions of other protection domains in the call stack are propagated to the Profile when a business method on the Profile Local interface is invoked. If the value of this attribute is "False", then the method in the Profile abstract class invoked as a result of a business method invoked on the Profile Local interface runs with an access control context that includes the protection domain(s) of the Profile Specification as well as the protection domains of any other classes in the call stack as prescribed by the Java security model, such as the SLEE Component that invoked the Profile Local interface method. If the value of the `isolate-security-permissions` attribute is "True", the SLEE automatically wraps the method invoked on the Profile abstract class in response to the Profile Local interface method invocation in an `AccessController.doPrivileged` block in order to isolate the security permissions of the invoked Profile, i.e. the security permissions of other protection domains in the call stack do not affect the invoked Profile.
- The `setProfileContext`, `unsetProfileContext`, `profilePostCreate`, `profileActivate`, `profilePassivate`, `profileLoad`, `profileStore`, and `profileRemove` methods defined in the `javax.slee.profile.Profile` interface can be invoked on a Profile object from an unpredictable call path. If any of these methods need to execute privileged code requiring security permissions over and above the standard set of permissions granted by the SLEE, the additional security permissions must be declared in the Profile Specification component jar's deployment descriptor and the relevant methods must wrap the privileged code in an `AccessController.doPrivileged` block to ensure that potentially more restrictive security permissions of other protection domains in the call stack do not prohibit the privileged code from being executed.
- The `profileInitialize` and `profileVerify` methods defined in the `javax.slee.profile.Profile` interface are invoked as a result of management operations

and therefore run with the default set of security permissions granted by the SLEE, plus any additional security permissions specified in the Profile Specification's component jar deployment descriptor.

The following security requirements apply to methods invoked on classes loaded from a Resource Adaptor component jar:

- All methods invoked by the SLEE on the `javax.slee.resource.ResourceAdaptor` and `javax.slee.resource.Marshaler` interfaces run with the default set of security permissions granted by the SLEE, plus any additional security permissions specified in the Resource Adaptor's component jar deployment descriptor.
- All methods that may be invoked by other SLEE Components such as SBB run with the set of security permissions that is the intersection of the permissions of all protection domains traversed by the current execution thread (up until any `AccessController.doPrivileged` invocation in the call stack). If any of these methods need to execute privileged code requiring security permissions over and above the standard set of permissions granted by the SLEE, the additional security permissions must be declared in the Resource Adaptor component jar's deployment descriptor and the relevant methods must wrap the privileged code in an `AccessController.doPrivileged` block to ensure that potentially more restrictive security permissions of other protection domains in the call stack do not prohibit the privileged code from being executed.

The following security requirements apply to methods invoked on classes loaded from a library component jar:

- All methods run with the set of permissions that is the intersection of the permissions of all protection domains traversed by the current execution thread (up until any `AccessController.doPrivileged` invocation in the call stack). If a library component method needs to execute privileged code requiring security permissions over and above the standard set of permissions granted by the SLEE, the additional security permissions must be declared in the library component jar's deployment descriptor and the method must wrap the privileged code in an `AccessController.doPrivileged` block to ensure that potentially more restrictive security permissions of other protection domains in the call stack do not prohibit the privileged code from being executed.

Some SLEE implementations may allow the Deployer to grant more, or fewer, permissions to the instances of other SLEE component classes than those specified above. Support for this is not required by the SLEE specification. SLEE components that rely on more or fewer permissions using non-specified SLEE features will not be portable across all SLEEs.

12.1.2 SLEE 1.1 APIs

The SLEE must implement the SLEE 1.1 interfaces as defined in this specification.

12.1.3 JNDI 1.2

Changed in 1.1: Various types of SLEE Components may access JNDI.

At the minimum, the SLEE must provide a JNDI API name space to the instances of each SBB, Profile Specification, and Resource Adaptor component's classes. For SBBs and Profile Specifications, the SLEE must make the name space available to an instance when the instance invokes the `javax.naming.InitialContext` default (no-arg) constructor. For Resource Adaptors, the SLEE provides a `javax.naming.Context` object to an instance when the instance invokes the `getNamingContext` method on the `javax.slee.resource.ResourceAdaptorContext` interface (refer Section 15.13).

The SLEE specification does not require that all the components deployed in a SLEE be presented with the same JNDI API name space. However, all the instances of the same component's classes must be presented with the same JNDI API name space.

12.2 Programming restrictions

Changed in 1.1: Various types of SLEE Components have different programming restrictions.

This section describes the programming restrictions that developers of various SLEE components must follow to ensure that the developed components are *portable* and can be deployed in any compliant SLEE. The restrictions apply to the implementation of the component methods. In general these programming restrictions apply regardless of security permissions.

12.2.1 Restrictions pertaining to all SLEE components

- A SLEE component must not attempt to query a class to obtain information about the declared members that are not otherwise accessible to the SLEE component because of the security rules of the Java language. The SLEE component must not attempt to use the Reflection API to access information that the security rules of the Java programming language make unavailable.

  Allowing the SLEE components to access information about other classes and to access the classes in a manner that is normally disallowed by the Java programming language could compromise security.

- A SLEE component must not attempt to create a class loader; set security manager; create a new security manager; stop the JVM; or change the input, output, and error streams. A SLEE component may obtain the class loader for its own classes for the purpose of loading any resource files stored in its component jar file only.

  These functions are reserved for the SLEE. Allowing the SLEE component to use these functions could compromise security and decrease the SLEE's ability to properly manage the runtime environment.

- A SLEE component must not attempt to obtain the security policy information for a particular code source.

  Allowing the SLEE component to access the security policy information would create a security hole.

- A SLEE component must not attempt to gain access to packages and classes that the usual rules of the Java programming language make unavailable to the SLEE component.

  This function is reserved for the SLEE. Allowing the SLEE component to perform this function would create a security hole.

- A SLEE component must not attempt to define a class in the `javax.slee` namespace.

  This function is reserved for the SLEE. Allowing the SLEE component to perform this function would create a security hole.

- A SLEE component must not attempt to access or modify the security configuration objects (Policy, Security, Provider, Signer, and Identity).

  These functions are reserved for the SLEE. Allowing the SLEE component to use these functions could compromise security.

- A SLEE component must not attempt to use the subclass and object substitution features of the Java Serialization Protocol.

  Allowing the SLEE component to use these functions could compromise security.

12.2.2 Restrictions pertaining to SBB, Profile, Event Type, and Resource Adaptor Type components

The following restrictions apply to these SLEE component types. They are in addition to the restrictions specified in section 12.2.1.

- These components must not use read/write static fields. Using read-only static fields is allowed. Therefore, it is recommended that all static fields in the classes of the SLEE component be declared as `final`.

  This rule is required to ensure consistent runtime semantics because while some SLEE implementations may use a single JVM to execute all instances of a SLEE component, others may distribute the instances across multiple JVMs.

- These components must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.

  Most servers do not allow direct interaction between an application program and a keyboard/display attached to the server system.

- These components should not use the `java.io` package to attempt to access files and directories in the file system.

  The file system APIs are not well-suited for SBB or Profile components to access data. SBB components should use a resource, such as JDBC, to store data, or may use the Profile mechanism.

- These components must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast.

  The SLEE architecture allows an instance of an SBB component class to be a network socket client, but it does not allow it to be a network server. Allowing the instance to become a network server would conflict with the basic function of the SBB component that is to process events.

- These components must not attempt to set the socket factory used by ServerSocket, Socket, or the stream handler factory used by URL.

  These networking functions are reserved for the SLEE. Allowing the components to use these functions could compromise security and decrease the SLEE's ability to properly manage the runtime environment.

- These components must not attempt to manage threads. Any of these components must not attempt to start, stop, suspend, or resume a thread; or to change a thread's priority or name. These components must not attempt to manage thread groups.

  Allowing these components to manage threads would decrease the SLEE's ability to properly manage the runtime environment.

- These components must not attempt to directly read or write a file descriptor.

  Allowing these components to read and write file descriptors directly could compromise security.

- These components must not attempt to set the context class loader.

  This function is reserved for the SLEE, Resource Adaptors, and Libraries. Allowing other SLEE component types to use these functions could compromise security and decrease the SLEE's ability to properly manage the runtime environment.

- These components must not attempt to load a native library.

  Allowing SBB, Profile, Resource Adaptor Type or Event Type components to load native code would create a security hole. This function is reserved for the SLEE, Resource Adaptors, and libraries.

12.2.3 Restrictions pertaining to SBB and Profile components

The following restriction applies to both SBB and Profile components. This in addition to the restrictions defined in sections 12.2.1 and 12.2.2.

- These components must not attempt to pass `this` as an argument or method result. Instead of writing code that uses the `this` reference and callback methods, an SBB Developer or Profile

Developer should attach to Activity Contexts to receive events or pass local object references between local methods invocations. *(Clarified in 1.1)*

- These components must not use thread synchronization primitives to synchronize execution of multiple instances. Instead they execute in a transaction, and therefore isolation is provided for their concurrency control.

  Synchronization would not work if the SLEE distributed multiple instances across multiple JVMs.

12.2.4 Restrictions pertaining to SBB components

The following restrictions apply to SBB components. These restrictions are in addition to the restrictions defined in sections 12.2.1, 12.2.2, and 12.2.3.

- An SBB component must not modify event objects passed to its event handler methods and must not hold a reference to the event object, directly or indirectly, after the event handler method returns.
- An SBB component must not modify event objects passed to its fire event methods.

12.2.5 Restrictions pertaining to Resource Adaptor and Library components

The following restriction applies to both Resource Adaptor and Library components. It is in addition to the restrictions defined in section 12.2.1.

- These components must not attempt to manage threads that they did not create. These components must not attempt to start, stop, suspend, or resume a thread they did not create; or to change a thread's priority or name if the thread was not created by the component. These components must not attempt to manage thread groups that they did not create.

Chapter 13 Facilities

Facilities are standard functional components provided by the SLEE. The SLEE specification defines the following standard Facilities:

- Timer Facility (see Section 13.1)
- Alarm Facility (see Section 13.2)
- Trace Facility (see Section 13.3)
- Activity Context Naming Facility (see Section 13.5)
- Profile Facility (see Section 13.6)
- Event Lookup Facility (see Section 13.7) *(Added in 1.1)*
- Service Lookup Facility (see Section 13.8) *(Added in 1.1)*

13.1 Timer Facility

SBB entities often need to perform periodic actions, or they need to initiate actions and then check at a later time to make sure they have been completed. The SLEE provides this function in the Timer Facility.

The Timer Facility manages a number of timers, each of which is independent. A timer may fire zero or more Timer Events.

The SLEE, together with its execution platform (OS, HW, … etc.) must ensure that logical time always advances.

13.1.1 `TimerID` interface

Every timer has a Timer identifier, which is included in every Timer Event that is fired by the Timer Facility for the timer. The Timer identifier is also used to cancel a timer. The Java interface of a `TimerID` object, which encapsulates a Timer identifier, is as follows.

```
package javax.slee.facilities;

public interface TimerID {
        public int hashCode();
        public boolean equals(Object obj);
        public String toString();
}
```

The SLEE provides the actual concrete implementation class of `TimerID` objects returned by the Timer Facility. This implementation class must meet the following requirements:

- Be serializable.
  This requirement allows SBB objects to store `TimerID` objects in their CMP fields.
- Implement `hashCode`.
  The actual implementation class of a `TimerID` object must override the `Object.hashCode` method. The new `hashCode` method does not need to provide hash codes that are unique but does need to provide hash codes that are well distributed. When different `TimerID` objects evaluate to the same hash code they are uniquely identified by their `equals` method. See `java.lang.Object` for more information.
- Implement `equals`.
  The actual implementation class of a `TimerID` object must override the `Object.equals` method. The new `equals` method must ensure that the implementation of the method is reflexive, symmetric, transitive and for any non-null value `X`, `X.equals(null)` returns false. See `java.lang.Object` for more information.

- Implement `toString`.
  The actual implementation class of a `TimerID` object must override the `Object.toString` method. The new `toString` method must provide a human readable representation of the `TimerID` object. If two `TimerID` objects represent two different Timer identifiers, the `toString` method must return two distinct string representations, i.e. `string1.equals(string2)` returns false. If two `TimerID` objects represent the same Timer identifier, the `toString` method must return the same string representation, i.e. `string1.equals(string2)` returns true.

13.1.2 `TimerFacility` interface

Changed in 1.1: JNDI location constant, and a method to retrieve the ActivityContext which a timer was set on were added.

SBB objects access the Timer Facility through a `TimerFacility` object that implements the `TimerFacility` interface. A `TimerFacility` object can be found at a SLEE specification defined location in every SBB's component environment.

The `TimerFacility` interface is as follows:

```
package javax.slee.facilities;

import javax.slee.Address;
import javax.slee.ActivityContextInterface;
import javax.slee.TransactionRolledbackLocalException;

public interface TimerFacility {
        // JNDI location constant
        public static final String JNDI_NAME =
                "java:comp/env/slee/facilities/timer";

        public TimerID setTimer(ActivityContextInterface activity,
                                Address address,
                                long startTime,
                                TimerOptions options)
                throws  NullPointerException,
                        IllegalArgumentException,
                        TransactionRolledbackLocalException,
                        FacilityException;
        public TimerID setTimer(ActivityContextInterface activity,
                                Address address,
                                long startTime,
                                long period,
                                int numRepetitions,
                                TimerOptions options)
                throws  NullPointerException,
                        IllegalArgumentException,
                        TransactionRolledbackLocalException,
                        FacilityException;
        public void cancelTimer(TimerID timerID)
                throws  NullPointerException,
                        TransactionRolledbackLocalException,
                        FacilityException;
        public ActivityContextInterface getActivityContextInterface(TimerID timerID)
                throws  NullPointerException,
                        TransactionRequiredLocalException,
                        FacilityException;
        public long getResolution()
                throws  FacilityException;
        public long getDefaultTimeout()
                throws  FacilityException;
}
```

- The `JNDI_NAME` constant. *(Added in 1.1)*
  This constant specifies the JNDI location where a `TimerFacility` object may be located by an SBB component in its component environment.
- All methods of the `TimerFacility` interface, except for the `getResolution` method and the `getDefaultTimeout` method, are required transactional methods. The `getResolution` method and the `getDefaultTimeout` method are non-transactional.
- The SLEE provides a concrete class implementing the `TimerFacility` interface.
- The methods of this interface throw the `javax.slee.facilities.FacilityException` if the requested operation cannot be completed because of a system-level failure.

13.1.2.1 `setTimer` methods

The `TimerFacility` interface defines the methods used to set and cancel timers. Timers are set with the `setTimer` methods. Each successful invocation of a `setTimer` method creates a new timer that has a unique Timer identifier. This Timer identifier is returned to the caller using a `TimerID` object.

The `setTimer` methods take an Activity Context Interface object as an argument. Each of these methods creates a timer that fires Timer Events on the specified Activity Context. If the Timer Facility creates the timer successfully, the Timer Facility will reference the provided Activity Context (see Section 7.1.4).

- The Timer Facility releases this reference when the timer is cancelled or after the timer has fired all its Timer Events. In addition, this Facility also releases all its references to an Activity Context after the SLEE has delivered an Activity End Event to all SBB entities that are attached to the Activity Context and can receive the Activity End Event (see Section 8.6.5). This helps the SLEE reclaim Activity Contexts.
- This reference may prevent the Activity Context from being reclaimed, especially if the Activity object that is encapsulated by the Activity Context is not ended implicitly by invoking a method, or by external events, such as disconnecting from a connection. For example, if the Activity Context of a `NullActivity` object is referenced and the `endActivity` method of the `NullActivity` object is never invoked, then the Activity Context will never be reclaimed if the timer causing the reference to be held never terminates to release the reference. *(Clarified in 1.1)*

Since an event is always fired on an Activity Context and with an optional default address (see Section 8.2.1), the second argument of the `setTimer` methods is an `Address` object. This argument provides the default address and is provided to the SLEE event router when the Timer Facility fires a Timer Event for the timer. Section 8.6 discusses how the default address provided to the event router influences event routing.

Each timer has a start time, i.e. the time at which the Timer Facility fires the first Timer Event for this timer.

A timer can be a periodic timer. The Timer Facility fires Timer Events periodically for a periodic timer. The Timer Facility fires the first Timer Event for the periodic timer at the specified start time and periodically fires subsequent Timer Events for the periodic timer at a given interval (or period) thereafter. The Timer Facility fires Time Events for a periodic timer until the periodic timer is cancelled or until the specified number of repetitions has been reached.

The following rules apply to the arguments of the `setTimer` methods.

- The `activity` argument.
  This argument must be a valid Activity Context Interface object. It cannot be `null`.
- The `address` argument.
  This argument provides the default address and can be `null` if there is no default address.
- The `startTime` argument.
  This argument specifies the start time of the timer. The start time is specified as the number of

milliseconds since January 1, 1970, 00:00:00 GMT (same as `Date.getTime()` and `System.currentTimeMillis()`). The start time of the timer is the time at which the Timer Facility fires the first Timer Event for the timer. If the specified start time is before the current time, then the start time is advanced to the current time.

- The `period` argument.
  This argument specifies the period of a repeating timer in milliseconds. It must be greater than zero.
- The `numRepetitions` argument.
  This argument specifies the number of repetitions for a periodic timer. If zero (0) is specified, the periodic timer will repeat infinitely.
- The `options` argument.
  This argument specifies the desired behavior of the timer (see Section 13.1.3). It cannot be `null`. The timeout value specified by the `options` argument must be less than or equal to the period specified by the `period` argument.

If any of the above arguments except `address` are `null`, the `setTimer` method throws a `java.lang.NullPointerException`. If the specified start time is less than zero, the specified period is zero or less, or the timeout value specified by the `options` argument is greater than the specified period, then the `setTimer` method throws a `java.lang.IllegalArgumentException`. If the `setTimer` methods are invoked without a valid transaction context and the transaction started for the invocation fails to commit, then these methods will throw a `javax.slee.TransactionRolledBackLocalException`. *(Clarified in 1.1)*

13.1.2.2 `cancelTimer` method

The `cancelTimer` method cancels the timer identified by the `timerID` argument (which is a `TimerID` object), A cancelled timer no longer generates new timer events. The SLEE is not required to remove timer events fired by the timer before it was cancelled but not yet delivered to SBB entities.

If the `timerID` argument is `null`, then this method throws a `java.lang.NullPointerException`. If the timer identified by this argument has already been cancelled or has fired all its Timer Events (some of these events may have not been delivered), then no action is taken.

If the `cancelTimer` method is invoked without a valid transaction context and the transaction started for the invocation fails to commit , then this method will throw a `javax.slee.TransactionRolledBackLocalException`.

13.1.2.3 `getActivityContextInterface` method

Added in 1.1

The `getActivityContext` method provides the `ActivityContextInterface` which the timer was set on. This method is a mandatory transacted method.

This method returns the generic `ActivityContextInterface` representing the Activity Context which the Timer was set on, or `null` if the Activity Context no longer exists. If the timer has expired or has been cancelled the SLEE will return the Activity Context which the timer was set on.

This method takes a single argument – the ID of a timer.

This method throws the following exceptions:

- A `java.lang.NullPointerException` is thrown if the argument is `null`.
- A `javax.slee.TransactionRequiredLocalException` is thrown if the method is invoked without a valid transaction context.

- A `javax.slee.facilities.FacilityException` is thrown if the SLEE is unable to perform this operation due to a system level failure.

13.1.2.4 `getResolution` method

Although the unit for the period of a periodic timer is specified in milliseconds, a Timer Facility implementation may be limited by the available clock resolution and/or scheduler implementation of the underlying platform. The approximate timer resolution of the Timer Facility can be obtained via the `getResolution` method. This allows SBBs to setup timers that take into account the timer resolution of the Timer Facility. The returned value is an estimate and may change over time to reflect observed SLEE and system behavior. Time resolution may be affected by system load.

13.1.2.5 `getDefaultTimeout` method

The `getDefaultTimeout` method returns the default timeout value (see Section 13.1.3.4). The default timeout value may be fixed by the SLEE implementation or may be configurable when the SLEE is installed or started.

13.1.3 `TimerOptions` objects

The `TimerOptions` object argument of a `setTimer` method specifies the desired behavior of the timer when the timer is created. The `TimerOptions` object is used to convey to the Timer Facility the desired behavior of the timer if there is a SLEE restart or when the Timer Facility cannot fire a Timer Event on time.

A `TimerOptions` object is used to specify the following options:

- The `persistent` option
- The `preserveMissed` option
- The `timeout` option.

The public interface of the `TimerOptions` class is as follows:

```
package javax.slee.facilities;

public class TimerOptions implements java.io.Serializable {
        // constructors
        public TimerOptions() { … }
        public TimerOptions(TimerOptions options) { … }
        public TimerOptions(long timeout, TimerPreserveMissed preserveMissed)
                throws IllegalArgumentException { … }

        // the preserveMissed option
        public TimerPreserveMissed getPreserveMissed() { … }
        public void setPreserveMissed(TimerPreserveMissed preserveMissed)
                throws NullPointerException {…}

        // the timeout value
        public long getTimeout() { … }
        public void setTimeout(long timeout)
                throws IllegalArgumentException { … }

        // deprecated methods
        public TimerOptions(boolean persistent, long timeout,
                            TimerPreserveMissed preserveMissed)
                throws IllegalArgumentException { … }
        public boolean isPersistent() { … }
        public void setPersistent(boolean persistent) { … }
}
```

The default constructor constructs a `TimerOptions` object with the `persistent` option set to `false`, the `preserveMissed` option set to `TimerPreserveMissed.LAST`, and the `timeout` option to zero (0).

13.1.3.1 `TimerPreserveMissed` objects

Changed in 1.1: String representation and fromString method added.

The `TimerPreserveMissed` class defines an enumerated type for the `preserveMissed` options supported by the Timer Facility. A singleton instance of each enumerated value is guaranteed (via an implementation of the `readResolve` method, see `java.io.Serializable` for more details), so that equality tests using == are always evaluated correctly. For convenience, an integer representation of `TimerPreserveMissed` objects is also available.

The following singleton `TimerPreserveMissed` objects define the valid arguments that may be provided to methods that take a `TimerPreserveMissed` object as argument and may be returned by methods that return `TimerPreserveMissed` objects.

- `ALL`
- `NONE`
- `LAST`

The public interface of the `TimerPreserveMissed` class is as follows:

```
package javax.slee.facilities;

public final class TimerPreserveMissed implements java.io.Serializable {
        // singletons
        public static final TimerPreserveMissed NONE …;
        public static final TimerPreserveMissed ALL …;
        public static final TimerPreserveMissed LAST …;

        // integer representation
        public static final int PRESERVE_NONE …;
        public static final int PRESERVE_ALL …;
        public static final int PRESERVE_LAST …;

        // string representation
        public static final String NONE_STRING = “None”;
        public static final String ALL_STRING = “All”;
        public static final String LAST_STRING = “Last”;

        // methods
        public boolean isNone() { … }
        public boolean isAll() { … }
        public boolean isLast() { … }

        public static TimerPreservedMissed fromString(String option)
                throws NullPointerException, IllegalArgumentException { … }
        public static TimerPreserveMissed fromInt(int option)
                throws IllegalArgumentException { … }
        public int toInt() { … }
        public String toString() { … }
}
```

- Each of the `is<`*`Option`*`>` methods determines if this `TimerPreserveMissed` object represents the *`<OPTION>`* option, and is equivalent to `(this ==` *`<OPTION>`*`)`. For example, the `isNone` method determines if this `TimerPreserveMissed` object represents the `NONE` option and is equivalent to `(this == NONE)`.
- The `fromInt` and `toInt` methods allow conversion between the `TimerPreserveMissed` object form and numeric form.

- The `fromInt` method throws a `java.lang.IllegalArgumentException` if the `option` argument is not one of the three integer representations.
- The `fromString` method allows conversion between `String` and `TimerPreserveMissed` forms. The conversion is case insensitive. This method throws a `java.lang.NullPointerException` if the argument is `null`. This method throws a `java.lang.IllegalArgumentException` if the argument is not an option known to this class.

13.1.3.2 `persistent` option

Deprecated in 1.1: This option has been deprecated. The semantics of persistent Timers were unclear as the definition of persistent implied by this option was inconsistent with the general use of the phrase "persistent" in the rest of the specification. If the SBB developer desires timers which survive the restart of the SLEE then the SBB developer should create such timers on receipt of a `ServiceStartedEvent` *in the SBB, rather than use the persistent option.*

This is a boolean option. If enabled (`true`), the timer is reinstated when the SLEE restarts after a complete shutdown (either due to catastrophic failure or operator shutdown). If disabled (`false`), the timer is cancelled and no further Timer Events will be fired by the timer.

Even though a timer is persistent, its Timer Events cannot be fired when the SLEE is not running. Depending on the `preserveMissed` option, Timer Events that would have been fired when the SLEE is not running may be fired when the SLEE is restarted.

13.1.3.3 `preserveMissed` option

This option specifies which and whether Timer Events will be fired when the Timer Facility cannot fire one or more of a timer's Timer Events on time. A Timer Event is not on time (or is late) if the Timer Facility cannot fire the Timer Event by (scheduled time + timeout value) (see Section 13.1.3.4).

Under normal conditions and expected workload, a SLEE and its Timer Facility should be able to fire Timer Events on time. However, under overload conditions or when the SLEE is not running, the SLEE Timer Facility may not be able to fire Timer Events on time.

The following three behaviors can be specified:

- `ALL`
  This option tells the Timer Facility to fire all of the timer's Timer Events no matter how late they are. It is useful to application code that depends on the total number of Timer Events received.
- `NONE`
  This option tells the Timer Facility not to fire the timer's Timer Events that are late. It is useful to application code that does not care for late Timer Events.
- `LAST`
  This option tells the Timer Facility to fire only the timer's most recent Timer Event. The most recent Timer Event may be the last in a series of late Timer Events or it may be the most recent on-time event following a series of late Timer Events. The Timer Facility does not fire earlier unfired Timer Events that were late except the most recent scheduled Timer Event. This behavior is useful to application code that would like to be notified when there is at least one late Timer Event but does not care to receive all outstanding late Timer Events.

The `TimerOptions` class will throw a `java.lang.NullPointerException` if a `null` value is provided to its `setPreserveMissed` or constructor methods.

13.1.3.4 The `timeout` option

This option specifies a timeout value in milliseconds. The Timer Facility uses this value to determine if a Timer Event is considered to be late or on time. A Timer Event is late if the Timer Facility cannot fire the

Timer Event by (scheduled time + timeout value). A Timer Event that is fired on time may not be delivered on time to an SBB entity because the Timer Event may be delayed in the SLEE's event queue or delayed by SBB entities with higher priorities.

For a periodic timer, the specified timeout value must be no greater than the period (or repetition interval) of the periodic timer. A zero timeout value specifies the default timeout value (see Section 13.1.2.5) as the timeout value if the period is more than the default timeout value. Otherwise, a zero timeout value specifies the period as the timeout value.

The Timer Facility should adjust timeout values that are smaller than the current timer resolution to the current timer resolution, or the specified period (for periodic timers only), whichever is less. This avoids dropping Timer Events as a result of insufficient timer resolution (see Section 13.1.6).

The `TimerOptions` class will throw a `java.lang.IllegalArgumentException` if a negative timeout value is provided to its `setTimeout` or constructor methods. Furthermore, if the timeout value specified by the `options` argument to a `setTimer` method is more than the `period` argument of the `setTimer` method, the Timer Facility will also throw an `IllegalArgumentException`.

13.1.4 Delays in delivering Timer Events

The Timer Facility is only concerned with whether a Timer Event is fired on time. Although the Timer Facility may fire a Timer Event on time, an SBB entity may not receive the Timer Event on time because of delays in the SLEE event router or because a higher priority SBB entity may have taken a long time to process the Timer Event.

It is possible to have multiple Timer Events belonging to the same timer queued in the SLEE's event queue for delivery. In some cases, these Timer Events may not be delivered in a timely manner. The SLEE Timer Facility is not aware of this condition and does not provide an option to collapse multiple outstanding Timer Events in the SLEE's event queue (similar to the `TimerPreservedMissed.LAST` option). This option was considered but not adopted because this option would require the Timer Facility to keep track of fired Timer Events. This would complicate implementations of the Timer Facility and consume more system resources.

13.1.5 Timer Events

The `event-definition` element for Timer Events is as follows:

```
<event-definition>
…
        <event-type-name> javax.slee.facilities.TimerEvent </event-type-name>
        <event-type-vendor> javax.slee </event-type-vendor>
        <event-type-version> 1.0 </event-type-version>
        <event-class-name> javax.slee.facilities.TimerEvent </event-class-name>
</event-definition>
```

The `TimerEvent` interface is as follows:

```
package javax.slee.facilities;

public interface TimerEvent {
        public TimerID getTimerID();
        public long getScheduledTime();
        public long getExpiryTime();
        public long getPeriod();
        public int getNumRepetitions();
        public int getRemainingRepetitions();
        public int getMissedRepetitions();
}
```

Each `TimerEvent` object contains:

- A `TimerID` object attribute.
  This attribute contains the Timer identifier returned by the `setTimer` method used to create the timer.

- A `scheduledTime` attribute.
  This attribute contains the time that the SLEE Timer Facility was scheduled to fire the Timer Event, also known as the scheduled time of the Timer Event. This is the time at which the Timer Event is expected to fire (assuming no latencies, i.e. a perfect system).
  - For a non-periodic timer or the first Timer Event of a periodic timer, the scheduled time of the Timer Event is the start time specified by the `startTime` argument of the `setTimer` method.
  - For subsequent Timer Events of a periodic timer, the scheduled time of the Timer Event is the scheduled time of the previous repetition plus the period of the timer (as specified by the `period` argument of the `setTimer` method).
- An `expiryTime` attribute.
  This attribute contains the time that the SLEE Timer Facility actually fired the Timer Event, also known as the actual expiry time of the Timer Event. This is the time at which the SLEE considers the event ready to enqueue the Timer Event into the SLEE's event queue.[26]
  - The Timer Facility provides an estimate of the expected maximum difference between scheduled and actual expiry times via the `getResolution` method of the `TimerFacility` interface. This estimate may change over time to reflect current SLEE behavior.
- A `period` attribute.
  This attribute contains the repetition period, as specified by the `period` argument of the `setTimer` method used to create the timer. A non-periodic timer has a `period` of `Long.MAX_VALUE`.
- A `numRepetitions` attribute.
  This attribute contains the number of repetitions requested, as specified by `numRepetitions` argument of the `setTimer` method used to create the timer. Its value is zero (0) for a periodic timer that repeats infinitely and one (1) for a non-repeating timer.
- A `remainingRepetitions` attribute.
  This attribute contains the number of outstanding repetitions, i.e. the potential number of Timer Events that the Timer Facility will fire in the future for this timer. Its value is `Integer.MAX_VALUE` for an infinitely repeating periodic timer and zero (0) for a non-periodic timer. Otherwise, its value is `numRepetitions` minus the repetition number of this Timer Event for a finite repeating periodic timer. *(Clarified in 1.1)*
- A `missedRepetitions` attribute.
  This attribute contains the number of missed repetitions between the last Timer Event and this Timer Event.

13.1.6 Desired Timer Facility behavior and Timer Facility pseudo code

The Timer Facility implementation must implement the behavior described below.

- The Timer Facility should aim to fire Timer Events with an actual expiry time as close as the scheduled time as practical.
- The Timer Facility may fire Timer Events with an actual expiry time that is before the scheduled time. However it should not fire timer events with an actual expiry time that is less than (scheduled time – timer resolution[27]). As timer resolution may change over time, users of the Timer Facility cannot rely on this behavior being strictly followed.

---

[26] The Timer Facility may not actually enqueue the Timer Event if it is marked as late or discarded under the `TimerPreserveMissed.NONE` or `TimerPreserveMissed.LAST` options.

[27] The `getResolution` method of the `TimerFacility` interface returns the timer resolution.

- The Timer Facility must consider all Timer Events where actual expiry time is greater than (scheduled time + timeout[28]) as late when timeout is greater than timer resolution. Late Timer Events are handled according to the `preserveMissed` option in the `TimerOptions` object originally passed to the `setTimer` method.

The following pseudo code illustrates the desired behavior of the Timer Facility. The following pseudo code is periodically executed for each active timer.

```
// timer                        is the timer object currently being examined.
// timer.scheduleTime           is time that the Timer Event is scheduled to fire.
// timer.timeout                is timeout for this timer (from TimerOptions).
// timer.numRepetitions         is the total repetitions for this timer, 0 if infinite,
//                              1 if non-periodic.
// timer.remainingRepetiton     is the remaining repetition count, initially Long.MAX_VALUE
//                              for infinite periodic timers, timer.numRepetitions
//                              otherwise.
// timer.period                 is the timer period (Long.MAX_VALUE if non-periodic).
// timer.missed                 is the counter of undelivered late events.
//
// Tres                         is the current timer resolution.
// Tsys                         is the current system time.
// Tdto                         is the current default time out
// Tfire                        is the "fire before" time; all Timer Events scheduled
//                              before this time are candidates for firing.
//
// Tsys <= Tfire <= Tsys + Tres
// Tres <= Tdto
//
// Timer Events fired are given with scheduled and actual expiry times, and for periodic
// timers, number of missed repetitions. Other attributes of the Timer Event can
// be inferred from current timer state.

if TimerOptions.preserveMissed == TimerPreserveMissed.ALL then
        while timer.scheduleTime < Tfire and timer.remainingRepetitions > 0 do
                fire event with (
                        event.scheduledTime = timer.scheduleTime,
                        event.expiryTime = Tsys,
                        event.missedRepetitions = 0)
                if timer.numRepetitions != 0 then
                        timer.remainingRepetitions -= 1
                                endif
                timer.scheduleTime += timer.period
        endwhile
else
        if timer.timeout == 0 then
                timeout = Tdto
        else
                timeout = timer.timeout
        endif
        timeout = minimum(maximum(timeout, Tres), timer.period)
        if TimerOptions.getPreserveMissed == TimerPreserveMissed.NONE then
                while timer.scheduleTime < Tfire and timer.remainingRepetitions > 0 do
                        if timer.remainingRepetitions == 1 and timeout < Tres then
                                // period goes to infinity for last Timer Event,
                                // need to change timeout value for cases
                                // influenced by period
                                timeout = Tres
                        endif
                        if Tsys <= timer.scheduleTime + timeout then
                                fire event with (
                                        event.scheduledTime = timer.scheduleTime,
                                        event.expiryTime = Tsys,
                                        event.missedRepetitions = timer.missed)
                                timer.missed = 0
```

[28] The timeout value is specified by the `timeout` attribute of the `TimerOptions` object originally passed to the `setTimer` method.

```
                        else
                                timer.missed += 1
                        endif
                        if timer.numRepetitions != 0 then
                                timer.remainingRepetitions -= 1
                                                        endif
                        timer.scheduleTime += timer.period
                endwhile
        else if TimerOptions.getPreserveMissed == TimerPreserveMissed.LAST then
                // Count missed events.
                // Preserve the last missed event for firing in the second loop.
                while timer.scheduleTime + period < Tfire and
                      Tsys > timer.scheduleTime + timeout and
                      timer.remainingRepetitions > 1 do
                        timer.missed += 1
                        if timer.numRepetitions != 0 then
                                timer.remainingRepetitions -= 1

                        endif
                        timer.scheduleTime += timer.period
                endwhile
                // Fire the last missed event preserved above (if any) and
                // any on-time events.
                while timer.scheduleTime < Tfire and timer.remainingRepetitions > 0 do
                        fire event with (
                                event.scheduled = timer.scheduleTime,
                                event.expiryTime = Tsys,
                                event.missedRepetitions = timer.missed)
                        timer.missed = 0
                        if timer.numRepetitions != 0 then
                                timer.remainingRepetitions -= 1
                        endif
                        timer.scheduleTime += timer.period
                endwhile
        endif
endif
if timer.numRepetitions != 0 and timer.remainingRepetitions == 0 then
cancel timer
endif
```

13.2 Alarm Facility

Changed in 1.1: Alarms in 1.0 were stateless. Alarms in 1.1 are stateful and have a lifecycle (i.e. they are raised and cleared). Each alarm has an identifier. New management operations have been defined the `javax.slee.management.AlarmMBean` *interface which allow alarms to be queried and cleared. The* `Level` *class has been deprecated and replaced by the* `AlarmLevel` *class in the Alarm Facility. The* `AlarmLevel` *class provides levels which match those defined by ITU X.733. The 1.0 APIs have been deprecated. Additionally various SLEE components may use the Alarm Facility.*

The Alarm Facility is used by SBBs, Resource Adaptors, and Profiles to request the SLEE to raise or clear alarms. If a request is made to raise an alarm and the identified alarm has not already been raised, the alarm is raised and a corresponding alarm notification is generated by the AlarmMBean. If a request is made to clear an alarm and the identified alarm is currently raised, the alarm is cleared and a corresponding alarm notification is generated by the AlarmMBean. Requests to raise an alarm that is already raised or to clear an alarm that is not currently raised cause no further action in the SLEE, ie. notifications are not generated in this case.

Alarm notifications are intended for consumption by management clients external to the SLEE. For example, these management clients may be a network management console or a management policy engine. In any case, the management client is responsible for registering to receive alarm notifications generated by the Alarm Facility through the external management interface of the Alarm Facility (see Section 14.14). The management client may optionally provide notification filters so that only the alarm notifications that the management client would like to receive are transmitted to the management client. These notification

filters reduce alarm related network traffic and the processing that the management client has to perform to filter unwanted alarm notifications.

The external management interface of the Alarm Facility is defined by the `AlarmMBean` interface. Each alarm notification generated by the Alarm Facility is emitted through an `AlarmMBean` object as an `AlarmNotification` object. See Section 14.14 for detailed description of the `AlarmMBean` interface and the `AlarmNotification` class.

The Alarm Facility may be used by other alarm sources within the SLEE to generate alarm notifications. These alarm sources include the various subsystems and Facilities of the SLEE. The SLEE specification does not define the interface that these alarm sources use to emit alarm notifications. However, each alarm notification is emitted through the same `AlarmMBean` object and as an `AlarmNotification` object. Each `AlarmNotification` object contains a `NotificationSource` object (see Section 14.16.1) that identifies the alarm source.

13.2.1 Alarms and alarm identifiers

Added in 1.1.

Alarms are either raised or clear. When raised the alarm level specifies the severity of the alarm. The Java type for an alarm is `javax.slee.management.Alarm`.

Every alarm has a notification source. The notification source identifies the component that raised the alarm. This may be either an SBB component and Service ID pair, a resource adaptor entity, a Profile Table, or a SLEE implementation defined internal sub-system.

Every alarm has an alarm type. The Java type for an alarm type is `java.lang.String`. The Alarm type allows the user of the Alarm Facility to provide classification information about the alarm messages emitted. The management client can use this classification information to filter the alarm notifications. Examples of alarm types could be “Database Connection Failed”, or “Registration Missing”. The same alarm type can be used by different notification sources.

Every alarm has an instance ID. The instance ID allows the user of the Alarm Facility to provide additional information about the particular alarm, within the classification. For example if the alarm type was “Database Connection Failed” the instance ID might indicate which database server has failed. For the example of the “Registration Missing” alarm type, the identifier of the user might be supplied as the instance ID.

An alarm is considered the same as another alarm if its notification source, alarm type and instance ID are the same.

Each *raised alarm* is uniquely identified by an alarm identifier generated by the SLEE. The Java type for an alarm identifier is `java.lang.String`. Each alarm identifier is associated with an alarm, and uniquely identifies the particular occurrence of the alarm being raised. For example if we consider an alarm which is raised, cleared and then raised again the two different periods where the alarm is raised are distinguished by different alarm identifiers.

13.2.2 `AlarmFacility` interface

Changed in 1.1: JNDI location constant added.

SBB, Profile, and Resource Adaptor objects access the Alarm Facility through an `AlarmFacility` object that implements the `AlarmFacility` interface. An `AlarmFacility` object can be found at a SLEE specification defined location in every SBB and Profile component environment and is also available to each resource adaptor entity via its `ResourceAdaptorContext`. An `AlarmFacility` object automatically provides notification source information to the SLEE when methods on the `AlarmFacility` interface are invoked.

The `AlarmFacility` interface is as follows:

```
package javax.slee.facilities;

import javax.slee.ComponentID;
```

```
import javax.slee.UnrecognizedAlarmException;
import javax.slee.UnrecognizedComponentException;

public interface AlarmFacility {
        // JNDI location constant
        public static final String JNDI_NAME =
                "java:comp/env/slee/facilities/alarm";

        // methods
        public String raiseAlarm(String alarmType, String instanceID,
                                 AlarmLevel alarmLevel, String message)
                throws  NullPointerException,
                        IllegalArgumentException,
                        FacilityException;
        public String raiseAlarm(String alarmType, String instanceID,
                                 AlarmLevel alarmLevel, String message,
                                 Throwable cause)
                throws  NullPointerException,
                        IllegalArgumentException,
                        FacilityException;
        public boolean clearAlarm(String alarmID)
                throws  NullPointerException,
                        FacilityException;
        public int clearAlarms(String alarmType)
                throws  NullPointerException,
                        FacilityException;
         public int clearAlarms()
                throws  FacilityException;

        // deprecated methods
        public void createAlarm(ComponentID alarmSource, Level alarmLevel,
                                String alarmType, String message, long timestamp)
                throws  NullPointerException,
                        IllegalArgumentException,
                        UnrecognizedComponentException,
                        FacilityException;
        public void createAlarm(ComponentID alarmSource,Level alarmLevel,
                                String alarmType, String message, Throwable cause,
                                long timestamp)
                throws  NullPointerException,
                        IllegalArgumentException,
                        UnrecognizedComponentException,
                        FacilityException;
}
```

- The `JNDI_NAME` constant. *(Added in 1.1)*
  This constant specifies the JNDI location where an `AlarmFacility` object may be located by an SBB or Resource Adaptor component in its component environment.
- All methods of the `AlarmFacility` interface are non-transactional methods.
- The SLEE provides the concrete implementation class of an `AlarmFacility` object. This implementation class implements the `AlarmFacility` interface.

13.2.2.1 `raiseAlarm` methods

Added in 1.1.

There are two `raiseAlarm` methods. These methods are used to request that an alarm is raised. An alarm will be raised if there is no existing alarm with matching identifying attributes raised. The alarm will remain active until it is cleared, either via a call to any of the `clearAlarm` methods on this interface, or by management client use of the `javax.slee.management.AlarmMBean` object. When an alarm is raised an alarm notification is emitted by the Alarm MBean.

The identifying attributes of an alarm are the notification source, the alarm type, and the instance ID.

The following arguments may be passed to these methods:

- The `alarmType` argument.
  This argument specifies the type of alarm being generated. It will be included in the `AlarmNotification` object emitted by the `AlarmMBean` object for the alarm notification. A management client can use the alarm type of the `AlarmNotification` object to filter the alarm notification.
- The `instanceID` argument.

  This argument specifies the particular instance of the alarm type that is occuring.
- The `alarmlevel` argument.
  This argument takes an `AlarmLevel` object that specifies the alarm level of the alarm notification. All the `AlarmLevel` objects defined in Section 13.2.3 can be specified except the `AlarmLevel.CLEAR` object.
- The `message` argument.
  This argument specifies the message that will be placed into the `message` attribute of the `AlarmNotification` object emitted by the `AlarmMBean` object for the alarm notification if the alarm is raised as a result of this method. The `AlarmNotification` object inherits its `message` attribute from the standard JMX `Notification` class.
- The optional `cause` argument.
  This argument is typically used to propagate an exception in the alarm notification.

The `raiseAlarm` method returns a `String` value which is used to identify the raised alarm. If an alarm with the same identifying attributes (notification source, alarm type, instanceID) is already raised then this method has no effect, and the identifier of the existing raised alarm is returned. If no such alarm is raised, then a new alarm is raised, the SLEE's `AlarmMBean` object emits an alarm notification, and a SLEE-generated alarm identifier for the new alarm is returned. The `raiseAlarm` method throws a `java.lang.NullPointerException` if any of the arguments, except `cause`, are `null`. It throws a `java.lang.IllegalArgumentException` if the specified alarm level is `AlarmLevel.CLEAR`. It throws the `javax.slee.facilities.FacilityException` if the requested operation cannot be completed because of a system-level failure.

13.2.2.2 `clearAlarm` method

Added in 1.1.

This method clears an alarm identified by the argument. If the identified alarm is not active this method returns `false`. If the identified alarm is active the alarm is cleared and the method returns `true`. This method only clears alarms associated with the notification source of the alarm facility object. If the alarm is cleared, the SLEE's `AlarmMBean` object emits an alarm notification for the alarm with the alarm level set to `AlarmLevel.CLEAR`.

This method takes the following argument:

- The `alarmID` argument.
  This argument specifies the alarm identifier of the alarm being cleared.

The `clearAlarm` method throws a `java.lang.NullPointerException` if the `alarmID` argument is `null`. It throws the `javax.slee.facilities.FacilityException` if the alarm can not be cleared because of a system-level failure.

13.2.2.3 `clearAlarms(String)` method

Added in 1.1.

This method requests that all alarms of a certain alarm type are cleared. If no alarms of this type are active this method returns silently. The return value of this method is the number of alarms that were cleared. This method only clears alarms associated with the notification source of the alarm facility object. If one or

more alarms are cleared by this method, the SLEE's `AlarmMBean` object emits an alarm notification for each cleared alarm with the alarm level set to `AlarmLevel.CLEAR`.

This method takes the following argument:

- The `alarmType` argument.
  This argument specifies the alarm type of the alarms to be cleared.

The `clearAlarms(String)` method throws a `java.lang.NullPointerException` if the `alarmID` argument is `null`. It throws the `javax.slee.facilities.FacilityException` if the alarms can not be cleared because of a system-level failure.

13.2.2.4 `clearAlarms()` method

Added in 1.1.

This method requests that all alarms belonging to the notification source associated with the `Alarm-Facility` object are cleared. The return value of this method is the number of alarms that were cleared. If one or more alarms are cleared by this method, the SLEE's `AlarmMBean` object emits an alarm notification for each cleared alarm with the alarm level set to `AlarmLevel.CLEAR`.

The `clearAlarms()` method throws a `javax.slee.facilities.FacilityException` if the alarms can not be cleared because of a system-level failure.

13.2.2.5 `createAlarm` methods

Deprecated in 1.1: The `createAlarm` methods have been deprecated. They have been replaced with the methods above that provide both stateful alarm management and a more general approach to specifying the source of an alarm. The SLEE specification strongly recommends the use of the new Alarm Facility methods in preference to these deprecated methods. The deprecated methods may be removed in a future version of this specification.

There are two `createAlarm` method. A `createAlarm` method is used to emit an alarm notification. It takes the following arguments:

- The `alarmSource` argument.
  This argument specifies the component that is emitting the alarm notification. It is propagated as the `alarmSource` attribute of an `AlarmNotification` object (see Section 14.14.2).
- The `alarmLevel` argument.
  This argument takes a `Level` object that specifies the alarm level of the alarm notification. All the `Level` objects defined in Section 13.4 can be specified except the `Level.OFF` object.
- The `alarmType` argument.
  This argument specifies the type of alarm being generated. It will be included in the `AlarmNo-tification` object emitted by the `AlarmMBean` object for the alarm notification. A management client can use the alarm type of the `AlarmNotification` object to filter the alarm notification.
- The `message` argument.
  This argument specifies the message that will be placed into the `message` attribute of the `AlarmNotification` object emitted by the `AlarmMBean` object for the alarm notification. The `AlarmNotification` object inherits its `message` attribute from the standard JMX `No-tification` class.
- The optional `cause` argument.
  This argument is typically used to propagate an exception in the alarm notification.
- The `timestamp` argument.
  This argument specifies the timestamp that will be placed into the `timestamp` attribute of the

`AlarmNotification` object emitted by the `AlarmMBean` object for the alarm notification. The `AlarmNotification` object inherits its `timestamp` attribute from the standard JMX `Notification` class.

The `createAlarm` method throws a `java.lang.NullPointerException` if any of the arguments are `null`. It throws a `java.lang.IllegalArgumentException` if the specified alarm level is `Level.OFF`. The `createAlarm` method throws a `javax.slee.UnrecognizedComponentException` if the `componentID` argument is invalid. The `createAlarm` method throws the `javax.slee.facilities.FacilityException` if the requested operation cannot be completed because of a system-level failure.

13.2.3 `AlarmLevel` objects

Added in 1.1.

The `AlarmLevel` class defines an enumerated type for the levels supported by the Alarm Facility. A singleton instance of each enumerated value is guaranteed (via an implementation of the `readResolve` method, see `java.io.Serializable` for more details), so that equality tests using == are always evaluated correctly. For convenience, an integer representation of `AlarmLevel` objects is also available.

The following singleton `AlarmLevel` objects define the valid arguments that may be provided to methods that take a `AlarmLevel` object as argument and may be returned by methods that return `AlarmLevel` objects. The alarm levels, in order of relative severity from highest to least, are:

- `CLEAR`
- `CRITICAL`
- `MAJOR`
- `WARNING`
- `INDETERMINATE`
- `MINOR`

The public interface of the `AlarmLevel` class is as follows:

```
package javax.slee.facilities;

public final class AlarmLevel implements java.io.Serializable {
        // singletons
        public static final AlarmLevel CLEAR …;
        public static final AlarmLevel CRITICAL …;
        public static final AlarmLevel MAJOR …;
        public static final AlarmLevel WARNING …;
        public static final AlarmLevel INDETERMINATE …;
        public static final AlarmLevel MINOR …;

        // integer representation
        public static final int LEVEL_CLEAR …;
        public static final int LEVEL_CRITICAL …;
        public static final int LEVEL_MAJOR …;
        public static final int LEVEL_WARNING …;
        public static final int LEVEL_INDETERMINATE …;
        public static final int LEVEL_MINOR …;

        // string representation
        public static final String CLEAR_STRING = “Clear”;
        public static final String CRITICAL_STRING = “Critical”;
        public static final String MAJOR_STRING = “Major”;
        public static final String WARNING_STRING = “Warning”;
        public static final String INDETERMINATE_STRING = “Indeterminate”;
        public static final String MINOR_STRING = “Minor”;
```

```
        // methods
        public boolean isClear() { … }
        public boolean isCritical() { … }
        public boolean isMajor() { … }
        public boolean isWarning() { … }
        public boolean isIndeterminate() { … }
        public boolean isMinor() { … }

        public boolean isHigherLevel(AlarmLevel other)
               throws NullPointerException { … }

        public String AlarmLevel fromString(String level)
               throws NullPointerException, IllegalArgumentException { … }
        public static AlarmLevel fromInt(int level)
               throws IllegalArgumentException { … }
        public int toInt() { … }
        public String toString() { … }
        public boolean equals(Object obj) { … }
        public int hashcode() { … }
}
```

- The `isHigherLevel` method returns true if the level represented by this `AlarmLevel` object is higher or more severe than the level specified by `AlarmLevel` object passed as the `other` argument. For the purpose of this comparison, `CLEAR` has a higher level than `CRITICAL`, i.e. when `CLEAR` is specified in an alarm filter (see Section 14.14.4), only alarm notifications that indicate an alarm has been cleared will pass through the filter. It throws a `java.lang.NullPointerException` if the `other` argument is `null`.
- Each of the `is<`*`Level`*`>` methods determines if this `AlarmLevel` object represents the `<`*`LEVEL`*`>` level, and is equivalent to `(this == <`*`LEVEL`*`>)`. For example, the `isMajor` method determines if this `AlarmLevel` object represents the `MAJOR` alarm level and is equivalent to `(this == MAJOR)`.
- The `fromInt` and `toInt` methods allow conversion between the `AlarmLevel` object form and numeric form.
- The `fromInt` method throws a `java.lang.IllegalArgumentException` if the `level` argument is not one of the six integer representations.
- The `fromString` method allows conversion between `String` and `AlarmLevel` forms. The conversion is case insensitive. This method throws a `java.lang.NullPointerException` if the argument is `null`. This method throws a `java.lang.IllegalArgumentException` if the argument is not an alarm level known to this class.

13.3 Trace Facility

Changed in 1.1: Various types of SLEE components may use the Trace Facility. The `TraceFacility` interface has been deprecated and replaced by the `Tracer` interface. The `Level` class has been deprecated and replaced by the `TraceLevel` class in the Trace Facility.

Notification sources such as SBBs, Resource Adaptors, Profiles, and SLEE internal components can use the Trace Facility to generate trace messages intended for consumption by external management clients, such as a network management console, a management policy engine, or an integrated development environment. Management clients register to receive trace messages generated by the Trace Facility through the external management interface of the Trace Facility (see Section 14.15). The management client can filter trace message in the following two ways:

- The external management interface of the Trace Facility provides a method to set the trace filter level for a notification source. Only trace messages that are emitted by a notification source with an equal or higher trace level than the notification source's trace filter level will be accepted and transmitted by the Trace Facility. The management client can use this management interface to set the trace filter levels for the various notification sources. Notification sources can obtain the trace

filter level set by the management client from the Trace Facility. This allows these notification sources to not generate trace messages whose trace level is below the trace filter level set by the management client.

- The management client can optionally use filters to filter trace messages so that only the trace messages that the management client would like to receive are transmitted to the management client. These filters reduce trace message related network traffic and the processing that the management client has to perform to filter unwanted trace messages.

The external management interface of the Trace Facility is defined by the `TraceMBean` interface. Each trace message transmitted by the Trace Facility is emitted through a `TraceMBean` object as a `TraceNotification` object. See Section 14.15 for detailed description of the `TraceMBean` interface and the `TraceNotification` class.

Within the SLEE, notification sources use a tracer to emit trace messages. A tracer is a named entity. Tracer names are case-sensitive and follow the Java hierarchical naming conventions. A tracer is considered to be an ancestor of another tracer if its name followed by a dot is a prefix of the descendant tracer's name. A tracer is considered to be a parent of a child tracer if there are no ancestors between itself and the descendant tracer. For example, the tracer named "`com`" is the parent tracer of the tracer named "`com.foo`" and an ancestor of the tracer named "`com.foo.bar`". Each tracer name can be divided into one or more name components. Each name component identifies the name of the tracer at a particular level in the hierarchy. For example, the tracer named "`com.foo.bar`" has three name components, "`com`", "`foo`", and "`bar`".

- Tracer name components must be at least one character in length. This means that the tracer name "`com.foo`" is a legal name, whereas "`com..foo`" is not, as the middle name component of this name is zero-length.
- The empty string, "", denotes the root tracer. The root tracer sits at the top of a tracer hierarchy and is the ancestor of all tracers. The root tracer always exists and always has an assigned trace level. The default trace level for all root tracers is `TraceLevel.INFO`.
- All tracers are implicitly associated with a notification source (see Section 14.16). The notification source identifies the object in the SLEE that is emitting the trace message and is included in trace notifications generated by the Trace MBean on behalf of the tracer.

13.3.1 `Tracer` interface

Added in 1.1.

A tracer is represented in the SLEE by the `Tracer` interface. Notification sources access the Tracer Facility through a `Tracer` object that implements the `Tracer` interface. A `Tracer` object can be obtained by SBBs via the `SbbContext` interface, by resource adaptor entities via the `ResourceAdaptorContext` interface, and by profiles via the `ProfileContext` interface.

The `Tracer` interface is as follows:

```
package javax.slee.facilities;

import javax.slee.facilities.FacilityException;

public interface Tracer {
       // methods
       public String getTracerName();
       public String getParentTracerName();

       public TraceLevel getTraceLevel()
              throws  FacilityException;
       public boolean isTraceable(TraceLevel traceLevel)
              throws  FacilityException;

       public void trace(TraceLevel traceLevel, String message)
              throws  NullPointerException,
```

```
                        IllegalArgumentException,
                        FacilityException;
        public void trace(TraceLevel traceLevel, String message, Throwable cause)
                throws  NullPointerException,
                        IllegalArgumentException,
                        FacilityException;

        // specific trace level utility methods
        public void severe(String message)
                throws  NullPointerException,
                        FacilityException;
        public void severe(String message, Throwable cause)
                throws  NullPointerException,
                        FacilityException;
        public boolean isSevereEnabled()
                throws  FacilityException;
        public void warning(String message)
                throws  NullPointerException,
                        FacilityException;
        public void warning(String message, Throwable cause)
                throws  NullPointerException,
                        FacilityException;
        public boolean isWarningEnabled()
                throws  FacilityException;
        public void info(String message)
                throws  NullPointerException,
                        FacilityException;
        public void info(String message, Throwable cause)
                throws  NullPointerException,
                        FacilityException;
        public boolean isInfoEnabled()
                throws  FacilityException;
        public void config(String message)
                throws  NullPointerException,
                        FacilityException;
        public void config(String message, Throwable cause)
                throws  NullPointerException,
                        FacilityException;
        public boolean isConfigEnabled()
                throws  FacilityException;
        public void fine(String message)
                throws  NullPointerException,
                        FacilityException;
        public void fine(String message, Throwable cause)
                throws  NullPointerException,
                        FacilityException;
        public boolean isFineEnabled()
                throws  FacilityException;
        public void finer(String message)
                throws  NullPointerException,
                        FacilityException;
        public void finer(String message, Throwable cause)
                throws  NullPointerException,
                        FacilityException;
        public boolean isFinerEnabled()
                throws  FacilityException;
        public void finest(java.lang.String message)
                throws  NullPointerException,
                        FacilityException;
        public void finest(String message, Throwable cause)
                throws  NullPointerException,
                        FacilityException;
        public boolean isFinestEnabled()
                throws  FacilityException;
}
```

- All methods of the `Tracer` interface are non-transactional methods. The effects of operations invoked on this facility occur regardless of the state or outcome of any enclosing transaction.

These methods are non-transactional because trace messages emitted from within a transaction should not be "discarded" if the transaction aborts. In fact, notification sources should use trace messages to document any unexpected occurrences, including the occurrences that lead to transaction rollbacks.

- The SLEE provides the concrete implementation class of a `Tracer` object. This implementation class implements the `Tracer` interface.
- The methods of this interface throw the `javax.slee.facilities.FacilityException` if the requested operation cannot be completed because of a system-level failure.
- A SLEE Component may store a `Tracer` object in an object instance field as the notification source the `Tracer` object is associated with does not change during the lifetime of the object.

13.3.1.1 `getTracerName` method

This method returns the tracer name of the tracer represented by this `Tracer` object.

13.3.1.2 `getParentTracerName` method

This method returns the name of the immediate parent tracer of the tracer represented by this `Tracer` object. If this `Tracer` object represents the root tracer, this method returns `null`.

13.3.1.3 `getTraceLevel` method

The `getTraceLevel` method returns the current trace level of the tracer and notification source represented by this `Tracer`. The trace level of a particular tracer is equal to the trace level assigned to it by an Administrator using a `TraceMBean` object (see Section 14.15.1). If a tracer has no assigned trace level, then the trace level of the tracer is inherited from its closest ancestor that does have a trace level assigned to it.

The `getTraceLevel` method throws a `javax.slee.facilities.FacilityException` if the trace level could not be obtained due to a system level failure.

13.3.1.4 `isTraceable` method

This method determines if the trace level currently set for the tracer and notification source represented by this `Tracer` object notification source is sufficient to cause a trace notification to be generated if a trace message was emitted at the trace level specified by the `traceLevel` argument. An invocation of this method can be used as a guard around a trace invocation to avoid unnecessary computation of a trace message that will ultimately be discarded.

The `this.isTraceable(fooLevel)` method equivalent to `!this.getTraceLevel().isHigherLevel(fooLevel)`.

13.3.1.5 `trace` methods

There are two `trace` methods. A `trace` method is used to emit a trace message. It takes the following arguments:

- The `traceLevel` argument.
  This argument takes a `TraceLevel` object that specifies the trace level of the trace message. All the `TraceLevel` objects defined in Section 13.3.2 can be specified except the `TraceLevel.OFF` object. If the specified trace level is equal or above the trace filter level set, the Trace Facility will accept the trace message and propagate the trace message by emitting a `TraceNotification` object from a `TraceMBean` object. Otherwise, the specified trace level is below the trace filter level set and the Trace Facility will not propagate the trace message any further.

- The `message` argument.
  This argument specifies the message that will be placed into the `message` attribute of the `TraceNotification` object emitted by the `TraceMBean` object for the trace message. The `TraceNotification` object inherits its `message` attribute from the standard JMX `Notification` class.
- The optional `cause` argument.
  This argument is typically used to propagate an exception in the trace message.

The `trace` method throws a `java.lang.NullPointerException` if any of the arguments, except `cause`, are `null`. It throws a `java.lang.IllegalArgumentException` if the specified trace level is `TraceLevel.OFF`.

13.3.1.6 Trace level utility methods

The `Tracer` interface defines a set of utility methods for each defined trace level. These utility methods are the equivalent of calling the `trace` or `isTraceable` method with the trace level specified in the method name. For example:

```
isFinestEnabled() == isTraceable(TraceLevel.FINEST)
finest(fooMsg, fooCause) == trace(TraceLevel.FINEST, fooMsg, fooCause)
```

13.3.2 `TraceLevel` objects

Added in 1.1.

The `TraceLevel` class defines an enumerated type for the trace levels supported by the Trace Facility. A singleton instance of each enumerated value is guaranteed (via an implementation of the `readResolve` method, see `java.io.Serializable` for more details), so that equality tests using == are always evaluated correctly. For convenience, an integer representation of `TraceLevel` objects is also available.

The following singleton `TraceLevel` objects define the valid arguments that may be provided to methods that take a `TraceLevel` object as argument and may be returned by methods that return `TraceLevel` objects. The least severe trace levels are listed first.

- `FINEST`
- `FINER`
- `FINE`
- `CONFIG`
- `INFO`
- `WARNING`
- `SEVERE`
- `OFF`

The public interface of the `TraceLevel` class is as follows:

```
package javax.slee.facilities;

public final class TraceLevel implements java.io.Serializable {
       // singletons
       public static final TraceLevel FINEST …;
       public static final TraceLevel FINER …;
       public static final TraceLevel FINE …;
       public static final TraceLevel CONFIG …;
       public static final TraceLevel INFO …;
       public static final TraceLevel WARNING …;
       public static final TraceLevel SEVERE …;
       public static final TraceLevel OFF …;
```

```
        // integer representation
        public static final int LEVEL_FINEST …;
        public static final int LEVEL_FINER …;
        public static final int LEVEL_FINE …;
        public static final int LEVEL_CONFIG …;
        public static final int LEVEL_INFO …;
        public static final int LEVEL_WARNING …;
        public static final int LEVEL_SEVERE …;
        public static final int LEVEL_OFF …;

        // string representation
        public static final String FINEST_STRING = “Finest”;
        public static final String FINER_STRING = “Finer”;
        public static final String FINE_STRING = “Fine”;
        public static final String CONFIG_STRING = “Config”;
        public static final String INFO_STRING = “Info”;
        public static final String WARNING_STRING = “Warning”;
        public static final String SEVERE_STRING = “Severe”;
        public static final String OFF_STRING = “Off”;


        // methods
        public boolean isFinest() { … }
        public boolean isFiner() { … }
        public boolean isFine() { … }
        public boolean isConfig() { … }
        public boolean isInfo() { … }
        public boolean isWarning() { … }
        public boolean isSevere() { … }
        public boolean isOff() { … }

        public boolean isHigherLevel(TraceLevel other)
                throws NullPointerException { … }
        public static TraceLevel fromString(String level)
                throws NullPointerException, IllegalArgumentException { … }
        public static TraceLevel fromInt(int level)
                throws IllegalArgumentException { … }
        public int toInt() { … }
        public String toString() { … }
        public boolean equals(Object obj) { … }
        public int hashcode() { … }

}
```

- Management clients use the `TraceLevel.OFF` object to specify maximum filtering.
- The `isHigherLevel` method returns true if the level represented by this `TraceLevel` object is higher or more severe than the level specified by `TraceLevel` object passed as the `other` argument. For the purpose of this comparison, `OFF` has a higher level than `SEVERE`, i.e. when `OFF` is specified in a trace filter, no traces at a lower level than `OFF`, which is all traces, will be emitted. It throws a `java.lang.NullPointerException` if the `other` argument is `null`.
- Each of the `is<`*`Level`*`>` methods determines if this `TraceLevel` object represents the *<LEVEL>* level, and is equivalent to `(this ==` *<LEVEL>*`)`. For example, the `isSevere` method determines if this `TraceLevel` object represents the `SEVERE` trace level and is equivalent to `(this == SEVERE)`.
- The `fromInt` and `toInt` methods allow conversion between the `TraceLevel` object form and numeric form.
- The `fromInt` method throws a `java.lang.IllegalArgumentException` if the `level` argument is not one of the eight integer representations.
- The `fromString` method allows conversion between `String` and `TraceLevel` forms. The conversion is case insensitive. This method throws a `java.lang.NullPointerException`

if the argument is `null`. This method throws a `java.lang.IllegalArgumentException` if the argument is not a trace level known to this class.

13.3.3 `TraceFacility` interface

Deprecated in 1.1: The Trace Facility interface is deprecated. It has been replaced by the `Tracer` interface. It identified the source of trace messages using a `ComponentID` object and thus was not flexible enough to be used in the wider range of notification sources available in SLEE 1.1. The SLEE specification strongly recommends the use of the new Trace Facility API available via `Tracer` objects in preference to these deprecated methods. The deprecated methods may be removed in a future version of this specification.

SBB objects access the Trace Facility through a `TraceFacility` object that implements the `TraceFacility` interface. A `TraceFacility` object can be found at a SLEE specification defined location in every SBB's component environment.

The `TraceFacility` interface is as follows:

```
package javax.slee.facilities;

import javax.slee.ComponentID;
import javax.slee.UnrecognizedComponentException;

public interface TraceFacility {
       // deprecated methods
       public Level getTraceLevel(ComponentID componentID)
              throws  NullPointerException,
                      UnrecognizedComponentException,
                      FacilityException;
       public void createTrace(ComponentID componentID,
                               Level level,
                               String messageType,
                               String message,
                               long timestamp)
              throws  NullPointerException,
                      IllegalArgumentException,
                      UnrecognizedComponentException,
                      FacilityException;
       public void createTrace(ComponentID componentID,
                               Level level,
                               String messageType,
                               String message,
                               Throwable cause,
                               long timestamp)
              throws  NullPointerException,
                      IllegalArgumentException,
                      UnrecognizedComponentException,
                      FacilityException;
}
```

- All methods of the `TraceFacility` interface are non-transactional methods. The effects of operations invoked on this facility occur irregardless of the state or outcome of any enclosing transaction.

  These methods are non-transactional because trace messages emitted from within a transaction should not be "discarded" if the transaction aborts. In fact, SBBs should use trace messages to document any unexpected occurrences, including the occurrences that lead to transaction rollbacks.

- The SLEE provides the concrete implementation class of a `TraceFacility` object. This implementation class implements the `TraceFacility` interface.

- The methods of this interface throw the `javax.slee.facilities.FacilityException` if the requested operation cannot be completed because of a system-level failure.

13.3.3.1 `getTraceLevel` method

The `getTraceLevel` method takes the following argument:

The `componentID` argument.
This argument specifies the SLEE component whose trace filter level should be returned.

The `getTraceLevel` method throws a `java.lang.NullPointerException` if the `componentID` argument is `null`. The `getTraceLevel` method throws a `javax.slee.UnrecognizedComponentException` if the `componentID` argument does not specify a component recognized by the SLEE.

13.3.3.2 `createTrace` methods

There are two `createTrace` methods. A `createTrace` method is used to emit a trace message. It takes the following arguments:

- The `componentID` argument.
  This argument specifies the component that the trace message is emitted for. Typically, the caller of this method determines whether the trace message is emitted for a Service or an SBB by providing a `ServiceID` object or an `SbbID` object for this argument. The `SbbID` object that identifies an SBB and the `ServiceID` object that identifies a Service can be found in an `SbbContext` object. This argument is propagated as the `messageSource` attribute of a `TraceNotification` object (see Section 14.15.2).
- The `level` argument.
  This argument takes a `Level` object that specifies the trace level of the trace message. All the `Level` objects defined in Section 13.4 can be specified except the `Level.OFF` object. If the specified trace level is equal or above the trace filter level set for the component identified by the `componentID` argument, the Trace Facility will accept the trace message and propagate the trace message by emitting a `TraceNotification` object from a `TraceMBean` object. Otherwise, the specified trace level is below the trace filter level set for the component identified by the `componentID` argument, the Trace Facility will not propagate the trace message any further.
- The `messageType` argument.
  This argument specifies the message type that will be included in the `TraceNotification` object emitted by the `TraceMBean` object. A management client can use the message type of the `TraceNotification` object to filter the trace message.
- The `message` argument.
  This argument specifies the message that will be placed into the `message` attribute of the `TraceNotification` object emitted by the `TraceMBean` object for the trace message. The `TraceNotification` object inherits its `message` attribute from the standard JMX `Notification` class.
- The optional `cause` argument.
  This argument is typically used to propagate an exception in the trace message.
- The `timestamp` argument.
  This argument specifies the timestamp that will be placed into the `timestamp` attribute of the `TraceNotification` object emitted by the `TraceMBean` object for the trace message. The `TraceNotification` object inherits its `timestamp` attribute from the standard JMX `Notification` class.

The `createTrace` method throws a `java.lang.NullPointerException` if any of the arguments are `null`. It throws a `java.lang.IllegalArgumentException` if the specified trace level is `Level.OFF`. The `createTrace` method throws a `javax.slee.UnrecognizedComponent-Exception` if the `componentID` argument does not specify a component recognized by the SLEE.

13.4 `Level` objects

Deprecated in 1.1: The `Level` class has been deprecated. Alarm and trace levels have been split into the `AlarmLevel` and `TraceLevel` classes respectively. The SLEE specification discourages the use of the deprecated APIs that use the deprecated `Level` objects as the may be removed in a future version of this specification.

The `Level` class defines an enumerated type for the levels supported by the SLEE 1.0 Alarm Facility and the Trace Facility APIs. A singleton instance of each enumerated value is guaranteed (via an implementation of the `readResolve` method, see `java.io.Serializable` for more details), so that equality tests using == are always evaluated correctly. For convenience, an integer representation of `Level` objects is also available.

The following singleton `Level` objects define the valid arguments that may be provided to methods that take a `Level` object as argument and may be returned by methods that return `Level` objects. The least severe alarm or trace levels are listed first.

- `FINEST`
- `FINER`
- `FINE`
- `CONFIG`
- `INFO`
- `WARNING`
- `SEVERE`

The public interface of the `Level` class is as follows:

```
package javax.slee.facilities;

public final class Level implements java.io.Serializable {
        // singletons
        public static final Level FINEST …;
        public static final Level FINER …;
        public static final Level FINE …;
        public static final Level CONFIG …;
        public static final Level INFO …;
        public static final Level WARNING …;
        public static final Level SEVERE …;
        public static final Level OFF …;

        // integer representation
        public static final int LEVEL_FINEST …;
        public static final int LEVEL_FINER …;
        public static final int LEVEL_FINE …;
        public static final int LEVEL_CONFIG …;
        public static final int LEVEL_INFO …;
        public static final int LEVEL_WARNING …;
        public static final int LEVEL_SEVERE …;
        public static final int LEVEL_OFF …;

        // methods
        public boolean isFinest() { … }
        public boolean isFiner() { … }
        public boolean isFine() { … }
        public boolean isConfig() { … }
```

```
        public boolean isInfo() { … }
        public boolean isWarning() { … }
        public boolean isSevere() { … }
        public boolean isOff() { … }

        public boolean isHigherLevel(Level other)
               throws NullPointerException { … }
        public static Level fromInt(int level)
               throws IllegalArgumentException { … }
        public int toInt() { … }
        public String toString() { … }
}
```

- Management clients use the `Level.OFF` object to specify maximum filtering.
- The `isHigherLevel` method returns true if the level represented by this `Level` object is higher or more severe than the level specified by `Level` object passed as the `other` argument. For the purpose of this comparison, `OFF` has a higher level than `SEVERE`, i.e. when `OFF` is specified in an alarm filter (see Section 14.14.4), no alarms at a lower level than `OFF`, which is all alarms, will be emitted. It throws a `java.lang.NullPointerException` if the `other` argument is `null`.
- Each of the `is<`*`Level`*`>` methods determines if this `Level` object represents the `<`*`LEVEL`*`>` level, and is equivalent to `(this == <`*`LEVEL`*`>)`. For example, the `isSevere` method determines if this `Level` object represents the `SEVERE` alarm level and is equivalent to `(this == SEVERE)`.
- The `fromInt` and `toInt` methods allow conversion between the `Level` object form and numeric form.
- The `fromInt` method throws a `java.lang.IllegalArgumentException` if the `level` argument is not one of the eight integer representations.

13.5 Activity Context Naming Facility

The Activity Context Naming Facility provides a global flat namespace for naming Activity Contexts. It allows an SBB object to bind a name to an Activity Context, and other SBB objects to lookup the Activity Context by this name. An Activity Context may be bound to zero or more names.

This Facility allows multiple SBB entities that have been created in response to initial events on different Activities (thus have different Activity Contexts) to converge on a "shared" Activity Context, so that these SBB entities can signal each other by firing events on the shared Activity Context and share common state by reading and modifying attributes.

SBBs that use this Facility are responsible for choosing unique names to avoid unintentional name conflicts.

When an Activity Context is bound to at least one name, the Activity Context Naming Facility maintains a reference to the Activity Context. It releases this reference when the Activity Context is no longer bound to any names. This occurs when the `unbind` method of this Facility is invoked to unbind the only name bound to the Activity Context. In addition, this Facility also releases its reference to an Activity Context after the SLEE has delivered an Activity End Event to all SBB entities that are attached to the Activity Context and can receive the Activity End Event (see Section 8.6.5). This helps the SLEE reclaim Activity Contexts.

The reference to the Activity Context held by the Activity Context Naming Facility may prevent the Activity Context from being reclaimed, especially if the Activity object that is encapsulated by the Activity Context is not ended implicitly by invoking a method, or by external events, such as disconnecting from a connection. For example, if the Activity Context of a `NullActivity` object is referenced and the `endAc-`

`tivity` method of the `NullActivity` object is never invoked, then the Activity Context will never be reclaimed.

13.5.1 `ActivityContextNamingFacility` interface

Changed in 1.1: JNDI location constant added.

SBB objects access the Activity Context Naming Facility through an `ActivityContextNaming-Facility` object that implements the `ActivityContextNamingFacility` interface. An `Activ-ityContextNamingFacility` object can be found at a SLEE specification defined location in every SBB's component environment.

The `ActivityContextNamingFacility` interface is as follows:

```
package javax.slee.facilities;

import javax.slee.ActivityContextInterface;
import javax.slee.TransactionRequiredLocalException;

public interface ActivityContextNamingFacility {
        // JNDI location constant
        public static final String JNDI_NAME =
                "java:comp/env/slee/facilities/activitycontextnaming";

        public void bind(ActivityContextInterface activity, String name)
                throws  NullPointerException,
                        NameAlreadyBoundException,
                        IllegalArgumentException,
                        TransactionRequiredLocalException,
                        FacilityException;
        public void unbind(String name)
                throws  NullPointerException,
                        NameNotBoundException,
                        TransactionRequiredLocalException,
                        FacilityException;
        public ActivityContextInterface lookup(String name)
                throws  NullPointerException,
                        TransactionRequiredLocalException,
                        FacilityException;
}
```

- The `JNDI_NAME` constant. *(Added in 1.1)*
  This constant specifies the JNDI location where an `ActivityContextNamingFacility` object may be located by an SBB component in its component environment.
- All methods of the `ActivityContextNamingFacility` interface are mandatory transactional methods (see Section 9.6.1).
- The SLEE provides the concrete implementation class of an `ActivityContextNaming-Facility` object. This implementation class implements the `ActivityContextNaming-Facility` interface.
- The methods of this interface throw the `javax.slee.facilities.FacilityException` if the requested operation cannot be completed because of a system-level failure.

13.5.1.1 `bind` method

The `bind` method takes the following arguments:

- The `activity` argument.
  This argument specifies an Activity Context Interface object that represents the Activity Context that will be bound to a name.

- The `name` argument.
  This argument specifies the name that will be bound to the Activity Context specified by the `activity` argument. A zero-length string is not a valid argument.

The Activity Context Naming Facility references the specified Activity Context on successful completion of this method if the Activity Context Naming Facility does not already reference the specified Activity Context.

This method throws a `java.lang.NullPointerException` if the `activity` argument or the `name` argument is `null`. This method throws a `javax.slee.facilities.NameAlreadyBoundException` if the specified name is already bound (either to the same Activity Context or another Activity Context). This method throws a `java.lang.IllegalArgumentException` if the specified name is a zero-length string.

13.5.1.2 unbind method

The `unbind` method takes the following argument:

- The `name` argument.
  This argument specifies the name that will be unbound from the Activity Context.

The Activity Context Naming Facility releases its reference to the Activity Context bound to the specified name on successful completion of this method if the Activity Context is only bound to the specified name.

This method throws a `java.lang.NullPointerException` if the `name` argument is `null`. This method throws a `javax.slee.facilities.NameNotBoundException` if the specified name is not already bound to an Activity Context.

13.5.1.3 lookup method

The `lookup` method takes the following argument:

- The `name` argument.
  This argument specifies the name that will be used to lookup an Activity Context.

This method returns `null` if the name is not bound to an Activity Context. It throws a `NullPointerException` if the `name` argument is `null`. Otherwise, it returns a generic Activity Context Interface object that represents the Activity Context.

13.5.2 Conference bridge example

In this example, users of the conference bridge make a call to a conference service at an 800 number. After the caller is connected to the conference service, the caller provides a session number that specifies a conference session that the caller wishes to attend.

The hypothetical Activity objects for this example are the `Call` objects and the `ConferenceSession` objects. When a caller calls the conference service, the incoming call causes a `Call` object to be created and an event to be emitted from the `Call` object. This event is an initial event for a new root Foo Conference SBB entity of the Foo Conference Service.

This Foo Conference SBB entity obtains the session number from the caller and uses the session number to construct a unique name that represents the session, e.g. “com.foobar.FooConference.800:session=” + session number. It looks up this name in the Activity Context Naming Facility.

- If this name is not already bound to an Activity Context, then the Foo Conference SBB entity invokes the Conference Bridge resource to create a new conference session. The Conference Bridge resource returns new a `ConferenceSession` object that represents the conference session. The SBB entity obtains the Activity Context of the `ConferenceSession` object and invokes the Activity Context Naming Facility to bind the above unique name of the conference session to the Activity Context of the `ConferenceSession` object.

- If this name is already bound to an Activity Context, then the Activity Context Naming Facility returns the Activity Context bound to this name. This Activity Context should encapsulate the `ConferenceSession` object that represents the conference session specified by the session number. This conference session has been created earlier by another Foo Conference SBB entity.

In either case, the Foo Conference SBB entity connects the call to the conference session, and attaches itself to the Activity Context of the `ConferenceSession` object.

The Activity Context Naming Facility allows Foo Conference SBB entities to locate a "shared" Activity Context that represents a conference session, and to maintain shared state relating to the conference sessions, such as the total number of callers on the conference session and other similar statistics.

13.6 Profile Facility

Changed in 1.1: A method has been added to the Profile Facility to provide access to `ProfileTable` objects. Methods which return Profile CMP interfaces have been deprecated.

The Profile Facility allows SBBs to obtain the `ProfileTable` object or `ProfileTableActivity` object for a specified Profile Table.

13.6.1 `ProfileFacility` interface

Changed in 1.1: JNDI location constant added.

SBB objects access the Profile Facility through a `ProfileFacility` object that implements the `ProfileFacility` interface. A `ProfileFacility` object can be found at a SLEE specification defined location in every SBB's component environment.

The `ProfileFacility` interface is as follows:

```
package javax.slee.profile;

import java.util.Collection;
import javax.slee.TransactionRolledbackLocalException;
import javax.slee.facility.FacilityException;

public interface ProfileFacility {
	// JNDI location constant
	public static final String JNDI_NAME =
		"java:comp/env/slee/facilities/profile";

	// methods
	public ProfileTableActivity getProfileTableActivity(String profileTableName)
		throws	NullPointerException,
			UnrecognizedProfileTableNameException,
			TransactionRolledbackLocalException,
			FacilityException;
	public ProfileTable getProfileTable(String profileTableName)
		throws	NullPointerException,
			UnrecognizedProfileTableNameException,
			FacilityException;

	// deprecated methods
	public Collection getProfiles(String profileTableName)
		throws	NullPointerException,
			UnrecognizedProfileTableNameException,
			TransactionRolledbackLocalException,
			FacilityException;
	public Collection getProfilesByIndexedAttribute(String profileTableName,
				String attributeName, Object attributeValue)
		throws	NullPointerException,
			UnrecognizedProfileTableNameException,
			UnrecognizedAttributeException,
			AttributeNotIndexedException,
			AttributeTypeMismatchException,
			TransactionRolledbackLocalException,
```

```
                        FacilityException;
        public ProfileID getProfileByIndexedAttribute(String profileTableName,
                                String attributeName, Object attributeValue)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        UnrecognizedAttributeException,
                        AttributeNotIndexedException,
                        AttributeTypeMismatchException,
                        TransactionRolledbackLocalException,
                        FacilityException;
}
```

- The `JNDI_NAME` constant. *(Added in 1.1)*
  This constant specifies the JNDI location where a `ProfileFacility` object may be located by an SBB component in its component environment.
- All methods of the `ProfileFacility` interface, except the `getProfileTable` method, are required transactional methods (see Section 9.6.2). They throw a `javax.slee.TransactionRolledbackLocalException` if they are invoked without a valid transaction context and the transaction started for the invocation fails to commit.
- The SLEE provides the concrete implementation class of a `ProfileFacility` object. This implementation class implements the `ProfileFacility` interface.
- The methods of this interface throw the `javax.slee.facilities.FacilityException` if the requested operation cannot be completed because of a system-level failure.

13.6.1.1 `getProfileTableActivity` method

The `getProfileTableActivity` method returns the Activity object of the Profile Table specified by the `profileTableName` argument. If the `profileTableName` argument does not identify a Profile Table recognized by the SLEE, the method throws a `javax.slee.profile.UnrecognizedProfileTableNameException`. If the `profileTableName` argument is `null`, the method throws a `java.lang.NullPointerException`.

13.6.1.2 `getProfileTable` method

Added in 1.1.

The `getProfileTable` method returns the `ProfileTable` object of the Profile Table specified by the `profileTableName` argument. The returned object may be cast to the profile table interface defined by the Profile Specification the Profile Table was created from using the standard Java typecast mechanism. However, if the SBB has not declared a `profile-spec-ref` in its deployment descriptor for the Profile Specification of the Profile Table specified by the `profileTableName` argument, then the SBB may not have the necessary classes in its classloader to perform this typecast. In addition, for the same reasons, the SBB may not be able to interact with Profile Local Objects obtained from the returned `ProfileTable` object other than via the generic SLEE-defined `ProfileLocalInterface`.

If the `profileTableName` argument does not identify a Profile Table recognized by the SLEE, the method throws a `javax.slee.profile.UnrecognizedProfileTableNameException`. If the `profileTableName` argument is `null`, the method throws a `java.lang.NullPointerException`.

13.6.1.3 `getProfiles` method

Deprecated in 1.1: This method is deprecated. It is replaced with the `findAll` method on the `ProfileTable` interface, which returns a `Collection` of profile local interface objects (see Section 10.8). The SLEE specification strongly recommends the use of the `ProfileTable` interface in preference to this method as this method may be removed in a future version of this specification.

The `getProfiles` method returns a `Collection` object that contains the Profile identifiers of the Profiles in the Profile Table identified by the `profileTableName` argument.

- The default Profile of the specified Profile Table is not considered to be in the Profile Table and will not be included in the collection since the default Profile does not have a Profile identifier.
- An `Iterator` object can be obtained from the `Collection` object that can be used to iterate over the Profile identifiers (i.e. the `next` method of the `Iterator` object returns a `ProfileID` object) of the Profiles in the Profile Table.
- If the `profileTableName` argument does not identify a Profile Table recognized by the SLEE, the method throws a `javax.slee.profile.UnrecognizedProfileTableName-Exception`. If the `profileTableName` argument is `null`, the method throws a `java.lang.NullPointerException`.
- The `Collection` object, its `Iterator` objects, and the Profile identifiers obtained from these objects are only valid within the transaction that returned the `Collection` object.

13.6.1.4 `getProfilesByIndexedAttribute` method

Deprecated in 1.1: This method is deprecated. It is replaced with the `findProfilesByAttribute` method on the `ProfileTable` interface, which returns a `Collection` of profile local interface objects (see Section 10.8.1). The SLEE specification strongly recommends the use of the `ProfileTable` interface in preference to this method as this method may be removed in a future version of this specification.

The `getProfilesByIndexedAttribute` method returns a `Collection` object that contains the Profile identifiers of the Profiles in the Profile Table identified by the `profileTableName` argument that satisfies the search criteria specified by the `attributeName` and `attributeValue` arguments. The `attributeName` attribute identifies the indexed attribute searched and the `attributeValue` argument identifies the value to look for in the specified attribute. If this method is called against a Profile Table created from a SLEE 1.1 Profile Specification the SLEE should reject the invocation by throwing a `FacilityException`.

- If the Java type of the attribute specified by the `attributeName` argument is not an array type, a Profile satisfies the search criteria if the value stored in the specified attribute is equal to the value specified by the `attributeValue` argument. If the Java type of the indexed attribute is not a primitive Java type, the appropriate Java type of the `attributeValue` argument must be the same as the Java type of the indexed attribute. If the Java type of the indexed attribute is a primitive Java type, the appropriate Java type of the `attributeValue` argument must be the Java wrapper class for the primitive type of the indexed attribute.
- If the Java type of the attribute specified by the `attributeName` argument is an array type, a Profile satisfies the search criteria if any element of the array object stored in the specified attribute is equal to the value specified by the `attributeValue` argument. If the Java type of the array element is not a primitive Java type, the appropriate Java type of the `attributeValue` argument must be the same as the Java type of the array element. If the Java type of the array element is a primitive Java type, the appropriate Java type of the `attributeValue` argument must be the Java wrapper class for the primitive type of the array element.
- If the Java type of the `attributeValue` argument is not an appropriate type for the indexed attribute, then this method throws a `javax.slee.profile.AttributeTypeMismatch-Exception`.
- The default Profile of the specified Profile Table is not considered to be in the Profile Table and will not be included in the collection since the default Profile does not have a Profile identifier.
- An `Iterator` object can be obtained from the `Collection` object that can be used to iterate over the Profile identifiers (i.e. the `next` method of the `Iterator` object returns a `ProfileID` object) of the Profiles in the Profile Table.

- If the `profileTableName` argument does not identify a Profile Table recognized by the SLEE, the method throws a `javax.slee.profile.UnrecognizedProfileTableNameException`. If the `attributeName` argument does not identify an attribute of the Profile Table, then the method throws a `javax.slee.profile.UnrecognizedAttributeException`. If the `attributeName` argument identifies an attribute that is not indexed, then the method throws a `javax.slee.profile.AttributeNotIndexedException`. If any of the method's arguments are `null`, the method throws a `java.lang.NullPointerException`.
- The `Collection` object, its `Iterator` objects, and the Profile identifiers obtained from these objects are only valid within the transaction that returned the `Collection` object.

13.6.1.5 `getProfileByIndexedAttribute` method

Deprecated in 1.1: This method is deprecated. It is replaced with the `findProfileByAttribute` method on the `ProfileTable` interface, which returns a `Collection` of profile local interface objects (see Section 10.8.1). The SLEE specification strongly recommends the use of the `ProfileTable` interface in preference to this method as this method may be removed in a future version of this specification.

The `getProfileByIndexedAttribute` method has the same semantics as the `getProfilesByIndexedAttribute` method except that the Profile identifier of the first Profile that satisfies the search criteria is returned. (The SLEE can stop the search after the first match.) This method returns `null` if no Profile in the specified Profile Table satisfies the search criteria. If this method is called against a Profile Table created from a SLEE 1.1 Profile Specification the SLEE should reject the invocation by throwing a `FacilityException`.

13.7 Event Lookup Facility

Added in 1.1.
The Event Lookup Facility is used by Resource Adaptors to obtain information about the event types installed in the SLEE. The Event Lookup Facility can be used to convert from SLEE event type component identifiers to the `FireableEventType` objects that a Resource Adaptor uses to identify event types (see Section 15.9).

13.7.1 `FireableEventType` interface

The `FireableEventType` interface is defined as follows:

```
package javax.slee.resource;

import javax.slee.EventTypeID;

public interface FireableEventType {
        public EventTypeID getEventType();
        public String getEventClassName();
        public ClassLoader getEventClassLoader()
}
```

- All methods of the `FireableEventType` interface are non-transactional methods.
- The SLEE provides a concrete class implementing the `FireableEventType` interface.

13.7.1.1 `getEventType` method

This method returns the event type identifier of the event type described by the `FireableEventType` object.

13.7.1.2 `getEventClassName` method

This method returns the fully-qualified name of the event class for the event type described by the `FireableEventType` object. This is the class name specified in the event type's deployment descriptor.

13.7.1.3 `getEventClassLoader` method

This method returns a class loader for the event type described by the `FireableEventType` object. This class loader can get used to load the class named by the `getEventClassName` method and may be used by a Resource Adaptor's `Marshaler` object to deserialize a marshaled event object.

13.7.2 `EventLookupFacility` interface

Resource Adaptors access the Event Lookup Facility through an `EventLookupFacility` object that implements the `EventLookupFacility` interface. An `EventLookupFacility` object can be obtained by a resource adaptor entity via its `ResourceAdaptorContext`.

The `EventLookupFacility` interface is defined as follows:

```
package javax.slee.facilities;

import javax.slee.EventTypeID;
import javax.slee.UnrecognizedEventException;
import javax.slee.resource.FireableEventType;

public interface EventLookupFacility {
       public FireableEventType getFireableEventType(EventTypeID eventType)
              throws  NullPointerException,
                      UnrecognizedEventException,
                      FacilityException;
}
```

- The `getFireableEventType` method is a non-transactional method.
- The SLEE provides the concrete implementation class of an `EventLookupFacility` object. This implementation class implements the `EventLookupFacility` interface.

13.7.2.1 `getFireableEventType` method

This method returns a `FireableEventType` object for the event type identified by the `eventType` argument.

This method throws a `java.lang.NullPointerException` if the `eventType` argument is `null`. It throws a `javax.slee.UnrecognizedEventException` if the `eventType` argument does not identify an event type in the SLEE. An `UnrecognizedEventException` is also thrown if the `eventType` argument does not identify an event type that the Resource Adaptor may fire (as determined by the resource adaptor types referenced by the Resource Adaptor) unless event type checking has been disabled for the Resource Adaptor (see Section 15.10). It throws a `javax.slee.facilities.FacilityException` if the SLEE cannot return the `FireableEventType` object due to a system level failure.

13.8 Service Lookup Facility

Added in 1.1.
The Service Lookup Facility is used by Resource Adaptors to obtain information about the types of events a Service installed in the SLEE may receive. The Service Lookup Facility can be used to convert from SLEE Service component identifiers to the `ReceivableService` objects that a Resource Adaptor uses to identify the Service that is the target of an event being fired.

13.8.1 `ReceivableService` interface

The `ReceivableService` interface is defined as follows:

```
package javax.slee.resource;

import javax.slee.ServiceID;

public interface ReceivableService {
        public ServiceID getService();
        public ReceivableService.ReceivableEvent[] getReceivableEvents();

        public boolean equals(Object obj);
}
```

- All methods of the `ReceivableService` interface are non-transactional methods.
- The SLEE provides a concrete class implementing the `ReceivableService` interface.

13.8.1.1 `getService` method

This method returns the Service component identifier of the Service described by the `ReceivableService` object.

13.8.1.2 `getReceivableEvents` method

This method returns an array of `javax.slee.resource.ReceivableService.ReceivableEvent` objects. Each `ReceivableEvent` object identifies an event type that one or more SBBs in the Service has an event handler method for. If different SBBs in the Service have specified to receive the same event type but with different resource options then a `ReceivableEvent` object is included in the returned array for each unique resource option for that event type. It is left to the Resource Adaptor to resolve these conflicts. In the case where multiple SBBs in the Service can receive the same event type and specify the same (or no) resource option for that event type, then only one `ReceivableEvent` object is included in the returned array for that event type.

The returned array will only include information on the event types that the Resource Adaptor may fire (as determined by the resource adaptor types it implements) unless event type checking has been disabled for the Resource Adaptor (see Section 15.10). If event type checking is disabled then the returned array contains information about all event types that can be received by the Service.

13.8.1.3 `equals` method

The implementation class of a `ReceivableService` object must override the `Object.equals` method. Two `ReceivableService` objects must be considered equals if the Service component identifiers returned by their respective `getService` methods are equal.

13.8.2 `ReceivableService.ReceivableEvent` interface

A `ReceivableService.ReceivableEvent` object contains information about a single event type that can be received by a Service. The `ReceivableService.ReceivableEvent` interface is defined as follows:

```
package javax.slee.resource;

import javax.slee.EventTypeID;

public interface ReceivableService.ReceivableEvent {
        public EventTypeID getEventType();
        public String getResourceOption();
        public boolean isInitialEvent();
}
```

- All methods of the `ReceivableService.ReceivableEvent` interface are non-transactional methods.
- The SLEE provides a concrete class implementing the `ReceivableService.ReceivableEvent` interface.

13.8.2.1 `getEventType` method

This method returns the event type identifier of the event type described by the `ReceivableService.ReceivableEvent` object.

13.8.2.2 `getResourceOption` method

This method returns the resource option specified in the deployment descriptor of the SBB receiving the event type. If the SBB did not specify a resource option then this method returns `null`.

13.8.2.3 `isInitialEvent` method

This method returns `true` if and only if the event type is received by the root SBB of the Service and the root SBB has flagged the event type as an initial event type in its deployment descriptor.

13.8.3 `ServiceLookupFacility` interface

Resource Adaptors access the Service Lookup Facility through a `ServiceLookupFacility` object that implements the `ServiceLookupFacility` interface. A `ServiceLookupFacility` object can be obtained by a resource adaptor entity via its `ResourceAdaptorContext`.

The `ServiceLookupFacility` interface is defined as follows:

```
package javax.slee.facilities;

import javax.slee.ServiceID;
import javax.slee.UnrecognizedServiceException;
import javax.slee.resource.ReceivableService;

public interface ServiceLookupFacility {
        public ReceivableService getReceivableService(ServiceID service)
                throws  NullPointerException,
                        UnrecognizedServiceException
                        FacilityException;
}
```

- The `getReceivableService` method is a non-transactional method.
- The SLEE provides the concrete implementation class of an `ServiceLookupFacility` object. This implementation class implements the `ServiceLookupFacility` interface.

13.8.3.1 `getReceivableService` method

This method returns a `ReceivableService` object for the Service identified by the `service` argument.

This method throws a `java.lang.NullPointerException` if the `service` argument is `null`. It throws a `javax.slee.UnrecognizedServiceException` if the `service` argument does not identify a Service in the SLEE. It throws a `javax.slee.facilities.FacilityException` if the SLEE cannot return the `ReceivableService` object due to a system level failure.

13.9 JNDI names of SLEE Facilities

The following table lists the SLEE specification defined names of the objects of SLEE Facilities under `java:comp/env`. Facilities that are only available to Resource Adaptors have no defined JNDI name as JNDI is not specified for Resource Adaptors.

Objects	***Names***
`TimerFacility` object	slee/facilities/timer
`AlarmFacility` object	slee/facilities/alarm
`TraceFacility` object (deprecated in SLEE 1.1)	slee/facilities/trace
`ActivityContextNamingFacility` object	slee/facilities/activitycontextnaming
`ProfileFacility` object	slee/facilities/profile
`ProfileTableActivityContextInterfaceFactory` object	slee/facilities/profiletableactivitycontextinterfacefactory
`ServiceActivityFactory` object	slee/serviceactivity/activitycontextfactory

Chapter 14 Management

The SLEE specification provides management capabilities for deployable units, Services, Resource Adaptors, Profile Tables, Profiles, the Alarm Facility, and the Trace Facility.

A management client manages the SLEE by interacting with a set of SLEE specification defined MBean objects, as defined by the Java Management Extension (JMX) specification. The SLEE specification defines the MBean interface of these MBean objects and the requirements for the MBean classes that implement these MBean interfaces.

14.1 Motivation for JMX instrumentation

The JMX instrumentation model was chosen for the following reasons:

- It enables the SLEE to be managed without heavy investment.
  Implementing instrumentation that conforms to the JMX architecture is easy.
- It provides a scaleable management architecture.
  The component-based approach of JMX means that JMX solutions, including the SLEE and its management clients, can scale from small devices to large telecommunications switches and beyond.
- It integrates easily with existing management solutions.
  Systems instrumented using JMX can be managed through HTML browsers or by various management protocols such as SNMP, TMN and WBEM. JMX instrumentation is protocol independent.

14.2 Changes in 1.1

The version of JMX used is updated to 1.2.1. The JMX Remote API 1.0 (JSR 16) may be used by a SLEE. The textual description of many methods have been clarified.

Several new interfaces have been defined, and new methods have been added to existing interfaces.

Component Identifier and descriptor interfaces have been converted to classes.

JMX Object Names for all MBean interfaces have been defined.

JAIN SLEE 1.0 Management clients are source code compatible with 1.1. Therefore their source code does not have to be modified to use 1.1 however a recompile against the 1.1 API is necessary.

Each interface and class describes relevant changes which have been made in 1.1.

14.3 SLEE MBean interfaces and classes

The SLEE MBean interfaces are the MBean interfaces defined by the SLEE specification. The SLEE specification defines the following MBean interfaces in the `javax.slee.management` package:

- The `SleeManagementMBean` interface.
  The `SleeManagementMBean` interface defines the primary management interface of the SLEE. It also provides access to many of the other MBean interfaces (see Section 14.5).
- The `DeploymentMBean` interface.
  The `DeploymentMBean` interface defines the management interface for installing components into the SLEE, removing components from the SLEE, and interrogating deployed components (see Section 14.6).
- The `ServiceManagementMBean` interface.
  The `ServiceManagementMBean` interface defines the management interface for Services installed in the SLEE (see Section 14.7).

- The `ServiceUsageMBean` interface.
  The `ServiceUsageMBean` interface defines the management interface used to interact with the usage parameter sets of SBBs in a Service (see Section 14.8).
- The `ProfileProvisioningMBean` interface.
  The `ProfileProvisioningMBean` interface defines the management interface for managing provisioned data in the SLEE (see Section 14.10).
- The `ProfileTableUsageMBean` interface. *(Added in 1.1)*
  The `ProfileTableUsageMBean` interface defines the management interface used to interact with the usage parameter sets of Profile Tables (see Section 14.11).
- The `ResourceManagementMBean` interface. *(Added in 1.1)*
  The `ResourceManagementMBean` interface defines the management interface for Resource Adaptors installed in the SLEE (see Section 14.12).
- The `ResourceUsageMBean` interface. *(Added in 1.1)*
  The `ResourceUsageMBean` interface defines the management interface used to interact with the usage parameter sets of resource adaptor entities (see Section 14.13).
- The `AlarmMBean` interface.
  The `AlarmMBean` interface defines the management interface for the SLEE's Alarm Facility (see Section 14.14).
- The `TraceMBean` interface.
  The `TraceMBean` interface defines the management interface for the SLEE's Trace Facility (see Section 14.15).

The SLEE specification also defines the following base MBean interfaces which provide common functionality required for deployed components:

- The `javax.slee.profile.ProfileMBean` interface. *(Added in 1.1)*
  The `ProfileMBean` interface is extended and implemented by a Profile MBean class generated by the SLEE for a Profile Specification (see Section 10.26).
- The `javax.slee.usage.UsageMBean` and `javax.slee.usage.UsageNotificationManagerMBean` interfaces. *(Added in 1.1)*
  The `UsageMBean` and `UsageNotificationManagerMBean` interfaces are extended and implemented by a Usage MBean class and Usage Notification Manager MBean class generated by the SLEE for a usage parameters interface (see Section 14.9).

A SLEE MBean class is a class that implements a particular SLEE MBean interface in accordance with the JMX specification. For example the Service Management MBean class presents the `javax.slee.management.ServiceManagementMBean` interface to the MBean Server and implements the attributes and operations defined by this interface.

14.3.1 Implementation requirements for a SLEE MBean class

Changed in 1.1: The required version of JMX has been updated to 1.2.1. The Service Management MBean now emits service state change notifications.

The following are the requirements that must be considered in the implementation of a SLEE MBean class.

- The SLEE MBean class must be implemented in accordance with the JMX 1.2.1 specification.
- The SLEE MBean class may be implemented as any type of MBean supported by the JMX 1.2.1 specification, provided that it presents the corresponding SLEE MBean interface to the MBean Server.

- If a method defined in a SLEE MBean interface returns one or more Object Names of other SLEE MBean objects, the implementation of the method must make sure that the respective MBean objects are registered with the SLEE's MBean Server before returning.
- If the SLEE MBean class generates notifications, it must implement the JMX `javax.management.NotificationBroadcaster` interface. The SLEE MBean interface itself does not extend this interface directly as the `addNotificationListener` and `removeNotificationListener` methods of the `NotificationBroadcaster` interface should not be exposed as management operations that can be invoked by the management client. Notification listeners should be added or removed via the MBean Server.

  The implementation classes of the following SLEE MBean interfaces emit notifications and must implement the `NotificationBroadcaster` interface:

  - The `SleeManagementMBean` interface.
    A `SleeManagementMBean` object generates SLEE state change notifications (see Section 14.5.4).
  - The `ServiceManagementMBean` interface.
    A `ServiceManagementMBean` object generates service state change notifications (see Section 14.7.2).
  - The `ResourceManagementMBean` interface.
    A `ResourceManagementMBean` object generates resource adaptor entity state change notifications (see Section 14.12.3).
  - The `AlarmMBean` interface.
    An `AlarmMBean` object generates alarm notifications (see Section 14.14.3).
  - The `TraceMBean` interface.
    A `TraceMBean` object generates trace notifications (see Section 14.15.2).
  - The `UsageMBean` interface.
    A `UsageMBean` object generates usage notifications (see Section 14.9.10).

  Each of these implementation classes must implement the notification broadcaster semantics defined by the JMX specification. This includes the filtering semantics defined by the JMX `NotificationBroadcaster` and `NotificationFilter` interfaces.

14.4 Accessing the MBean Server and SLEE MBean objects

The SLEE specification mandates the use of the JMX 1.2.1 MLet (management applet) specification in order for management clients to gain access to the SLEE MBean Server and SLEE MBean objects.

14.4.1 Requirements of the SLEE Vendor

Changed in 1.1: A SLEE vendor may allow the use of JMX Remote API 1.0 (JSR 160)

The SLEE specification requires that a SLEE vendor implement necessary functionality to load and instantiate management clients implemented as MLet MBeans. For example, the `javax.management.loading.MLet` MBean can be used by a SLEE implementation to perform parsing of a MLet text file and instantiation of the MBeans defined in it, however this method of loading MLet MBeans is not prescriptive. The SLEE must ensure that the MBean Server that the MLets are registered with is the same MBean Server that the SLEE MBean objects are registered with (or a remote image of it) in order for the MLet to invoke the SLEE's management operations.

The SLEE specification does not strictly define when during a SLEE's lifetime management client MLets should be instantiated. However, the earliest that a SLEE implementation should instantiate MLets is after the initialization phase is complete and the SLEE is ready to accept management operations. MLets may be instantiated any time after this point at the discretion of the SLEE implementation.

A SLEE vendor may also optionally allow management clients to connect to the SLEE's MBean Server via a remote connection established in accordance with the JMX Remote API 1.0 (JSR 160).

14.4.2 Requirements of the Management Client

Changed in 1.1: The use of JMX 1.2.1 is mandated.

The general requirements of any management client that accesses the SLEE MBean objects are as follows:

- The management client may not hold any direct references to a SLEE MBean object. It may only reference a SLEE MBean object by its Object Name. In other words, it can only invoke a SLEE MBean object through a local or remote MBean Server, and it identifies the SLEE MBean object to be invoked by the Object Name of the SLEE MBean object. Interaction with a SLEE MBean via a proxy object such as that created by the `javax.management.MBeanServerInvocationHandler` class is acceptable also.
- Since the management client cannot hold a direct reference to a SLEE MBean object, it cannot directly add or remove itself as a notification listener to a SLEE MBean object that implements the `NotificationBroadcaster` interface. Rather, the management client registers and removes a listener by invoking the MBean Server's `addNotificationListener` and `removeNotificationListener` methods, passing to these methods the Object Name of the SLEE MBean object that the management client wants to begin or stop receiving notifications from.

There are two typical approaches to implementing a custom SLEE management client. The first approach is to implement the client as a MLet MBean. The second approach is to implement a SLEE management client using the JMX Remote API.

A SLEE management client implemented as an MLet MBean behaves as a JMX connector or protocol adaptor. This MLet is registered with the SLEE MBean Server and provides an adaptation layer between the SLEE management operations and the management client.

The requirements of a management client MLet MBean are:

- The MLet can be implemented as any type of MBean supported by the JMX 1.2.1 specification.
- The MLet should implement the `javax.management.MBeanRegistration` interface. The `preRegister` method defined in this interface provides the MLet with the MBean Server instance that the SLEE MBean objects are registered with.
- A suitable MLet text file or other documentation that provides the necessary codebase and instantiation class information should be provided with the MLet distribution.

A SLEE management client implemented using the JMX Remote API interacts with a remote image of the SLEE's MBean server. The remote MBean server forwards management client requests to the real MBean server running in the SLEE, thereby bypassing the need for the management client vendor to provide an MLet that provides protocol adaptation layer functionality. This implementation approach assumes that the SLEE implementation the management client is connecting to allow clients to connect via a JMX Remote API connection.

14.4.3 Accessing SLEE MBean objects

Changed in 1.1: SLEE-defined Object Names of JMX MBeans are now specified. New MBean interfaces have been defined in 1.1 and therefore this section has been updated accordingly.

Figure 19 illustrates the access paths to the Object Names of the various MBean objects that provide the external management interface of the SLEE.

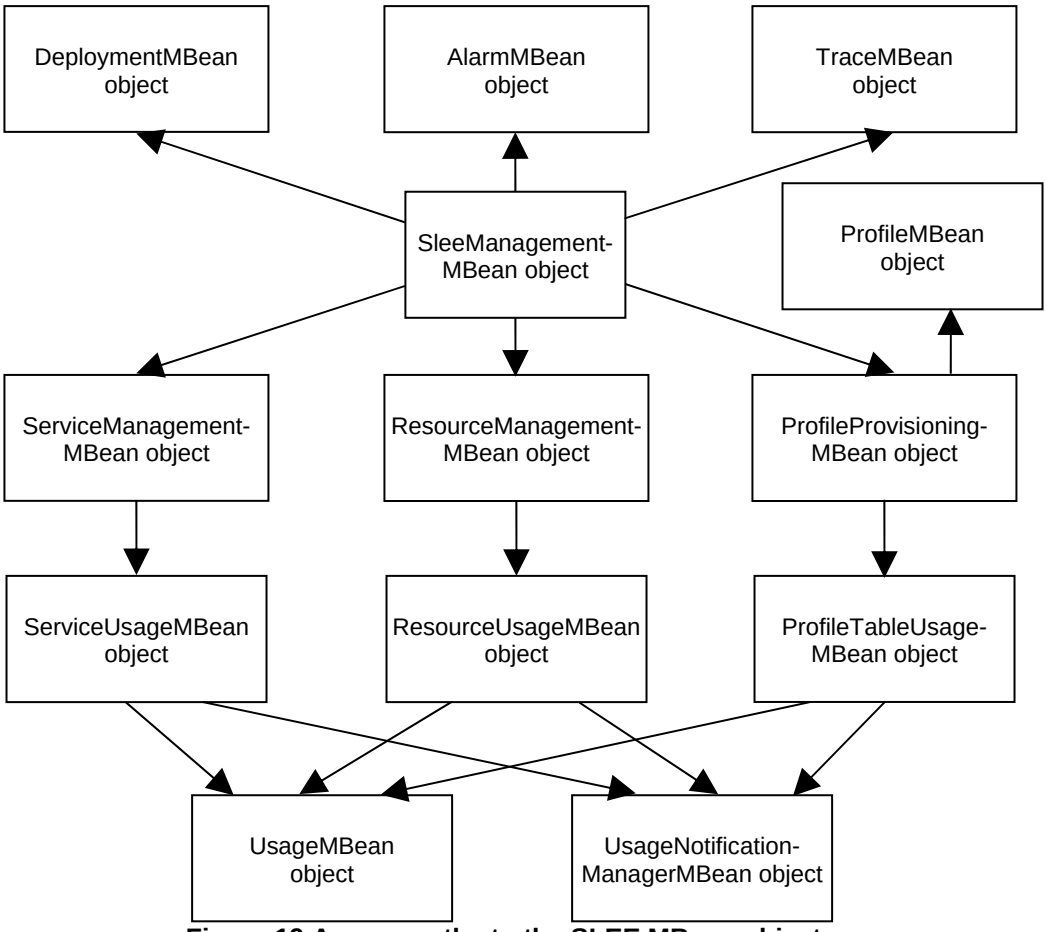

Figure 19 Access paths to the SLEE MBean objects

The SLEE specification defines the Object Name of every MBean object that the SLEE may make available to management clients. In the cases where more than one MBean object of a particular MBean type may exist, the SLEE specification defines the attributes of the MBean object's Object Name and the values those attributes must take.

Using the SLEE MBean Server, a management client can:

- obtain the names of other SLEE MBean objects (as shown in Figure 19)[29],
- invoke SLEE MBean objects,
- add and remove notification listeners to the SLEE MBean objects that generate notifications.

Deprecation note: The SLEE 1.0 specification defined a `SleeProviderFactory` *class that provided management clients with an object that returned the vendor-defined Object Name of the* `SleeManagementMBean` *object. The SLEE 1.1 specification formally defines the Object Names of all SLEE MBeans, thus the need for the* `SleeProviderFactory` *class has been deprecated. Refer Section 14.4.4 for more details.*

14.4.3.1 Object Name attribute values and quotations

Added in 1.1.

[29] This is an alternative option to manually constructing an MBean's Object Name from the defined rules.

The `equals` method of the JMX `ObjectName` class considers the following two Object Name strings to be different:

- "`javax.slee:type=ResourceUsage,raEntityName=FooResource`"
- "`javax.slee:type=ResourceUsage,raEntityName="FooResource"`"

Although otherwise identical, the quotes around the value of the `raEntityName` attribute cause the `ObjectName` class to consider the values to be different, and hence the name as a whole to be different.

For this reason, and in order to avoid incompatibilities, the SLEE specification requires certain MBean Object Name attribute values to be quoted. The `javax.management.ObjectName.quote(String)` method should be used to quote the following attribute values:

- For ProfileMBean Object Names:
  - The `profileTableName` and `profileName` attribute values.
- For Service Usage MBean Object Names:
  - The `serviceName`, `serviceVendor`, and `serviceVersion` attribute values.
- For Resource Usage MBean Object Names:
  - The `raEntityName` attribute value.
- For Usage MBean Object Names:
  - The `parameterSetName` attribute value.
  - The values of all attributes specific to a `NotificationSource` object, i.e. the `serviceName`, `serviceVendor`, `serviceVersion`, `sbbName`, `sbbVendor`, and `sbbVersion` attributes of `SbbNotification`, the `profileTableName` attribute of `ProfileTableNotification`, the `raEntityName` attribute of `ResourceAdaptorEntityNotification`, and the `subsystemName` attribute of `SubsystemNotification`.
- For Usage Notification Manager MBean Object Names:
  - The values of all attributes specific to a `NotificationSource` object, i.e. the `serviceName`, `serviceVendor`, `serviceVersion`, `sbbName`, `sbbVendor`, and `sbbVersion` attributes of `SbbNotification`, the `profileTableName` attribute of `ProfileTableNotification`, the `raEntityName` attribute of `ResourceAdaptorEntityNotification`, and the `subsystemName` attribute of `SubsystemNotification`.

14.4.4 SLEE Provider Factory

Changed in 1.1: Define the implications of defining the JMX Object Names of various specification defined MBeans.

The SLEE 1.0 specification did not define the Object Names of the SLEE MBeans, leaving the structure and format of these Object Names up to the SLEE vendor. In order to allow a management client to interact with the SLEE MBeans, the `SleeProviderFactory` class and `SleeProvider` interface were defined. Together these provided a management client with the Object Name of the `SleeManagementMBean` object, and from that MBean the Object Names of all other SLEE MBeans could be obtained (either directly or indirectly).

As the SLEE 1.1 specification defines the Object Names of all SLEE MBeans, the `SleeProviderFactory` class and `SleeProvider` interface have been deprecated. However a SLEE 1.1 compliant SLEE implementation must continue to support these interfaces. The SLEE 1.1 specification provides the `javax.slee.management.SleeProviderImpl` class as a default implementation of the `SleeProvider` interface which returns the SLEE 1.1-defined Object Name of the `SleeProviderMBean`

object. A 1.1-compliant SLEE must offer the `javax.slee.management.SleeProviderImpl` class, or an equivalent implementation class, as the `peerClassName` argument to the `SleeProvider-Factory` class methods.

14.4.4.1 `SleeProviderFactory` class

Deprecated in 1.1: This inteface has been deprecated. It is replaced by use of the Object Names defined in 1.1.

The `SleeProviderFactory` class provides static methods for obtaining an object instance of a SLEE vendor implemented concrete class that implements the `SleeProvider` interface. The `SleeProvid-erFactory` class is as follows:

```
package javax.slee.management;

public final class SleeProviderFactory {
       public static SleeProvider getSleeProvider(String peerClassName)
                       throws  NullPointerException,
                               PeerUnavailableException { … }
       public static SleeProvider getSleeProvider(String peerClassName,
                                                  ClassLoader classloader)
                       throws  NullPointerException,
                               PeerUnavailableException { … }
}
```

The `getSleeProvider` methods take the following argument:

- The `peerClassName` argument.
  This argument specifies the fully qualified class name of the concrete SLEE vendor implemented Java class that implements the `SleeProvider` interface.
  The SLEE 1.1 specification provides the `javax.slee.management.SleeProviderImpl` class which implements the `SleeProvider` interface to return the Object Name of the SLEE Management MBean (as defined by the SLEE 1.1 specification). The name of the `Slee-ProviderImpl` class can be used as a `peerClassName` value.

The second form of the `getSleeProvider` method also takes the following argument:

- The `classloader` argument.
  This argument specifies a class loader that should be used to load the class specified by the `peerClassName` argument.

These methods attempt to instantiate an object instance of the specified Java class, and, if successful, return the instantiated `SleeProvider` object.

These methods throw a `java.lang.NullPointerException` if `peerClassName` is `null`. In the second form, a `NullPointerException` is also thrown if `classloader` is `null`. It throws a `javax.slee.management.PeerUnavailableException` if the `peerClassName` argument does not identify a class that implements the `SleeProvider` interface (in some cases the specified class cannot be found, or the argument is not well-formed).

14.4.4.2 Vendor implemented `SleeProvider` concrete class

A SLEE vendor must provide at least one concrete class that implements the `SleeProvider` interface. The vendor implementation allows a management client MLet to obtain the Object Name of the `Slee-ManagementMBean`, which allows a management client to manage the SLEE.

Typically, a management client obtains an instance of this concrete class via the `getSleeProvider` method of the `SleeProviderFactory`. This concrete class must define a public no-argument constructor for use by the `getSleeProvider` method.

The `SleeProvider` interface is as follows:

```
package javax.slee.management;

import javax.management.ObjectName;

public interface SleeProvider {
       public ObjectName getSleeManagementMBean();
}
```

- The `getSleeManagementMBean` method.
  This method returns the Object Name of the `SleeManagementMBean` object.

14.4.4.3 SLEE implemented `SleeProvider` concrete class

Added in 1.1.

The SLEE 1.1 specification provides a default implementation of the `SleeProvider` class. The `SleeProviderImpl` class implements the `SleeProvider` interface and returns the Object Name of the SLEE Management MBean as defined by the SLEE 1.1 specification.

The `SleeProviderImpl` class is as follows:

```
package javax.slee.management;

import javax.management.ObjectName;

public final class SleeProviderImpl implements SleeProvider {
       public ObjectName getSleeManagementMBean() { … }
}
```

- The `getSleeManagementMBean` method.
  This method returns the Object Name of the `SleeManagementMBean` object, as defined by the `SleeManagementMBean` class' `OBJECT_NAME` attribute.

14.5 `SleeManagementMBean` interface

Changed in 1.1: The JMX Object Name of the `SleeManagementMBean` is now specified. New methods have been added to this interface.

The `SleeManagementMBean` interface defines the primary management interface of the SLEE. It can be used to:

- Change the operational state of the SLEE.
- Obtain the JMX Object Names of the `AlarmMBean`, `DeploymentMBean`, `ResourceManagementMBean`, `ServiceManagementMBean`, `TraceMBean`, and `ProfileProvisioningMBean` objects.
- Obtain the names of SLEE-vendor defined internal components or subsystems that may generate alarm, trace, or usage notifications. *(Added in 1.1)*
- Obtain usage information from SLEE-vendor defined internal components or subsystems. *(Added in 1.1)*

The Object Name of a `SleeManagementMBean` object is defined by the string "`javax.slee:name=SleeManagement`". *(Added in 1.1)*

Each time the operational state of the SLEE changes, the `SleeManagementMBean` object generates a SLEE state change notification. The `SleeStateChangeNotification` class defines the information encapsulated in each SLEE state change notification. The concrete class that implements the `SleeManagementMBean` interface must extend the `NotificationBroadcaster` interface even though the `SleeManagementMBean` interface does not extend the `NotificationBroadcaster` interface (see Section 14.3.1).

14.5.1 Operational states of the SLEE

Clarified in 1.1.

The following figure illustrates the operational states of the SLEE and permitted transitions between these states.

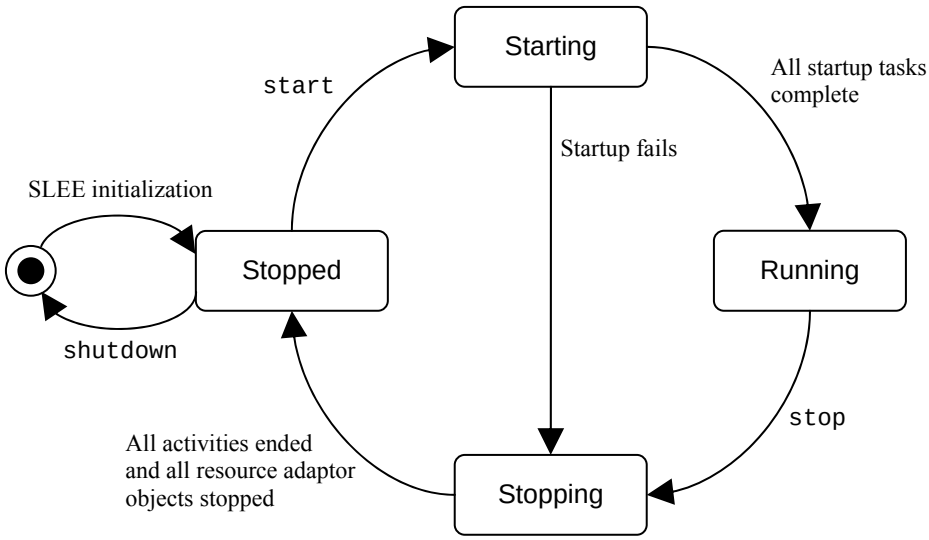

Figure 20 Operational states of the SLEE
All specified methods are on the `SleeManagementMBean` interface

- Stopped state.
  The SLEE environment is configured and initialized, ready to be started. This means resource adaptor objects for resource adaptor entities in the Active state are loaded and initialized, and SBBs corresponding to Services in the Active state are loaded and ready to be instantiated. However the entire event-driven subsystem is idle. Resource adaptor entities and the SLEE are not actively producing events, and the event router is not operating. SBB entities are not created in this state. *(Clarified in 1.1)*
  - Transitions:
    Stopped to Starting: The SLEE has no operations to execute
    Stopped to Does Not Exist: The SLEE process(es) shutdown and terminate gracefully.
- Starting state.
  Any vendor-specific starting state tasks may be performed here. Services in the Active state are made ready to receive events, but SBB entities are still not created in this state. The SLEE spontaneously moves out of this state when (a) startup tasks are complete, which causes transition to the Running state; (b) some startup task fails, which causes transition to the Stopping state. *(Clarified in 1.1)*
  - Transitions:
    Starting to Running: The event router is started. `ServiceActivity` objects are created for Services that are in the Active state, and a Service Started Event is fired on each of these `ServiceActivity` objects. Resource adaptor objects for resource adaptor entities that are Active transition to the Active state. The SLEE may also choose to create Profile Table Activity objects for each profile table that exists in the SLEE during this transition. *(Clarified in 1.1)*
    Starting to Stopping: The SLEE has no operations to execute.
- Running state.
  Resource adaptor objects that are in the Active (or Stopping) state are actively firing events as

conditions dictate. The SLEE may also be firing events. The event router is instantiating SBB entities and delivering events to them as required. *(Clarified in 1.1)*

- Transitions:
  Running to Stopping: Any `ServiceActivity` objects or `ProfileTableActivity` objects that exist are ended by the SLEE, and an Activity End Event is fired on each of them. Resource adaptor objects that are in the Active state transition to the Stopping state. *(Clarified in 1.1)*

- Stopping state.
  This state is identical to the Running state except no new Activity objects are accepted by the SLEE from resource adaptor objects, and no new Activity objects are created by the SLEE. If this state is reached from the Starting state, there will be no Activity objects and transition to the Stopped state should immediately occur. If this state is reached from the Running state, any existing Activity objects are allowed to end (subject to an arbitrary vendor-specified timeout). Once all Activity objects generated by a particular resource adaptor entity have ended, resource adaptor objects of that resource adaptor entity transition to the Inactive state. The SLEE transitions out of the Stopping state once all Activity objects have ended and all resource adaptor objects have transitioned to the Inactive state. *(Clarified in 1.1)*

  - Transitions:
    Stopping to Stopped: If the event router had been started, it is stopped. The implementation of a SLEE must ensure that shutdown cannot fail. For example, if a resource adaptor object fails to deactivate properly, the SLEE should just ignore any further events it produces.

The Stopped and Running states are stable, in that transitions to other states do not occur unless initiated by the Administrator. The Starting and Stopping states are transitional hence spontaneously move to another state as conditions dictate. *(Clarified in 1.1)*

14.5.1.1 Initialization and initial state

When a SLEE boots, it may exist in an indeterminate state until it is fully initialized. After initialization has completed, the SLEE enters the Stopped state. Management connections should only be permitted once the SLEE enters a valid state, therefore it is recommended that client connections be rejected while the SLEE is initializing. The simplest way to achieve this is to delay the instantiation of management client MLet MBeans until the very end of the initialization cycle. (*Clarified in 1.1)*

A SLEE vendor may choose to persist the operational state of the SLEE and re-enter that state when the SLEE server process(es) restart. For example, if a SLEE was in the Running state when it experienced a critical failure, on restart the SLEE may automatically attempt to re-enter the Running state and continue. In doing so however, the SLEE must migrate through the Stopped and Starting states as normal before re-entering the Running state (or Stopping state, if the restart fails).

14.5.1.2 Management operations and SLEE state

The SLEE state has no effect on the availability of any management MBean. A SLEE vendor may prescribe when management operations may be available, however they may only do so by limiting the availability of a SLEE MBean Server. If a SLEE instantiates and registers a management client MLet with an MBean Server, the SLEE vendor must ensure that the SLEE allows full and total access to its management subsystem via that MBean Server.

A SLEE vendor is free to make more than one SLEE MBean Server available for management clients to execute management operations. Where more than one SLEE MBean Server is available at a single point in time, it is the responsibility of the SLEE vendor to maintain a consistent management image across the multiple MBean Servers.

14.5.2 `SleeManagementMBean` interface

The `SleeManagementMBean` interface is as follows:

```
package javax.slee.management;

import javax.management.ObjectName;
import javax.slee.InvalidStateException;

public interface SleeManagementMBean {
	// JMX Object Name string of the SleeManagementMBean
	public static final String OBJECT_NAME
		= "javax.slee.management:name=SleeManagement";

	// JMX notification type of SLEE State Change notification
	public static final String SLEE_STATE_CHANGE_NOTIFICATION_TYPE =
		"javax.slee.management.sleestatechange";

	// get information about the SLEE implementation
	public String getSleeName();
	public String getSleeVendor();
	public String getSleeVersion();

	// get current operational state of the SLEE
	public SleeState getState()
		throws ManagementException;

	// change operational state of the SLEE
	public void start() throws InvalidStateException, ManagementException;
	public void stop() throws InvalidStateException, ManagementException;
	public void shutdown() throws InvalidStateException, ManagementException;

	// get the Object Names of the other SLEE MBean objects made accessible
	// by this SleeManagementMBean object
	public ObjectName getDeploymentMBean();
	public ObjectName getServiceManagementMBean();
	public ObjectName getProfileProvisioningMBean();
	public ObjectName getTraceMBean();
	public ObjectName getAlarmMBean();
	public ObjectName getResourceManagementMBean();

	// get usage information about internal component of the SLEE implementation
	public String[] getSubsystems()
		throws ManagementException;
	public boolean hasUsage(String subsystemName)
		throws  NullPointerException,
			UnrecognizedSubsystemName,
			ManagementException;
	public String[] getUsageParameterSets(String subsystemName)
		throws  NullPointerException,
			UnrecognizedSubsystemNameException,
			InvalidArgumentException,
			ManagementException;
	public ObjectName getUsageMBean(String subsystemName)
		throws  NullPointerException,
			UnrecognizedSubsystemNameException,
			InvalidArgumentException,
			ManagementException;
	public ObjectName getUsageMBean(String subsystemName, String paramSetName)
		throws  NullPointerException,
			UnrecognizedSubsystemNameException,
			UnrecognizedParameterSetNameException,
			InvalidArgumentException,
			ManagementException;
	public ObjectName getUsageNotificationManagerMBean(String subsystemName)
		throws  NullPointerException,
			UnrecognizedSubsystemNameException,
			InvalidArgumentException,
```

```
                    ManagementException;
}
```

- The `OBJECT_NAME` constant. *(Added in 1.1)*
  This constant defines the string form of the `SleeManagementMBean`'s JMX Object Name.
- The `SLEE_STATE_CHANGE_NOTIFICATION_TYPE` constant.
  This constant defines the value for the `type` attribute of `SleeStateChangeNotification`s emitted by a `SleeManagementMBean` object (refer Section 14.5.4).
- The `getSleeName` method.
  This method returns the name of the SLEE implementation. This may be a product name.
- The `getSleeVendor` method.
  This method returns the vendor of the SLEE implementation.
- The `getSleeVersion` method.
  This method returns the version of the SLEE implementation.
- The `getState` method.
  This method returns a `SleeState` object that indicates the current operational state of the SLEE.
  - It throws a `javax.slee.management.ManagementException` if the current operational state of the SLEE cannot be obtained due to a system-level failure.
- Methods that change the operational state of the SLEE, i.e. start and stop.
  - The `start` method.
    This method can only be called when the SLEE is in the Stopped state, and causes propagation to the Starting state
    - Conditions:
      Pre: `SleeManagementMBean.getState()` == `SleeState.STOPPED`
      Post: `SleeManagementMBean.getState()` == `SleeState.STARTING`
  - The `stop` method.
    This method can only be called when the SLEE is in the Running state, and causes propagation to the Stopping state.
    - Conditions:
      Pre: `SleeManagementMBean.getState()` == `SleeState.RUNNING`
      Post: `SleeManagementMBean.getState()` == `SleeState.STOPPING`
  - The `shutdown` method.
    This method can only be called when the SLEE is in the Stopped state, and causes all SLEE processes running under the same server image to shutdown and terminate in a graceful manner.
    - Conditions:
      Pre: `SleeManagementMBean.getState()` == `SleeState.STOPPED`
      Post: SLEE server processes no longer exist for the respective server image.

  The `start` and `stop` methods should not block. The operations the SLEE needs to perform in the transitional states, and further transitions to stable states, should be performed in a separate thread of execution to the thread that invoked the `start` or `stop` management operation.

  The `shutdown` method should never return, i.e. `System.exit()` should be invoked before control is returned to the caller.
  - The above methods may throw the following exceptions:

  - `javax.slee.InvalidStateException`.
    This exception is thrown if a requested change to the operational state of the SLEE is not permitted from the current operational state.
  - `javax.slee.management.ManagementException`.
    If the requested operation cannot be performed due to a system-level failure, then the method throws a `ManagementException`. The requested operation may be valid but the SLEE cannot perform the requested operation due to a system-level failure, or the SLEE cannot determine whether the requested operation is valid due to a system-level failure.

- Methods that get the Object Names of the SLEE MBean objects made accessible by this `SleeManagementMBean` object.
  - The `getDeploymentMBean` method.
  - The `getServiceManagementMBean` method.
  - The `getProfileProvisioningMBean` method.
  - The `getTraceMBean` method.
  - The `getAlarmMBean` method.
  - The `getResourceManagementMBean` method.
- Methods that are used to obtain information about the internal components or subsystems of the SLEE not defined by the SLEE specification. These methods are provided so that SLEE vendors have the option to expose usage and other information about subsystems specific to their SLEE implementation to management clients. *(Added in 1.1)*
  - The `getSubsystems` method.
    This method is used to determine the names of the internal components or subsystems provided by the SLEE implementation. Any vendor-defined subsystem that generates trace, alarm, or usage notifications should be named in the return value of this method. The SLEE specification does not define any restrictions on the format of subsystem names.
  - The `hasUsage` method.
    This method is used to determine whether the SLEE internal component or subsystem identified by the `subsystemName` parameter is providing usage information.
  - The `getUsageParameterSets` method.
    This method is used to obtain the names of the usage parameter sets of the SLEE internal component or subsystem identified by the `subsystemName` parameter.
    The SLEE specification does not provide an API for management clients to create or remove usage parameter sets for SLEE internal components or subsystems. The management of these usage parameter sets is the responsibility of the SLEE implementation alone.
  - The `getUsageMBean` methods.
    These methods are used to obtain a Usage MBean object for a SLEE internal component or subsystem that is providing usage information. The `subsystemName` parameter identifies the SLEE internal component or subsystem. The `paramSetName` parameter identifies a particular usage parameter set of the subsystem.

    The two-argument form of the `getUsageMBean` method may throw the following exception:

    - `javax.slee.usage.UnrecognizedUsageParameterSetNameException`.
      This exception is thrown if the usage parameter set named by the `paramSet-`

`Name` argument does not exist for the SLEE internal component or subsystem named by the `subsystemName` argument.

- The `getUsageNotificationManagerMBean` methods.
  This method is used to obtain a Usage Notification Manager MBean object for the SLEE internal component or subsystem named by the `subsystemName` parameter.
- The above methods may also throw the following exceptions:
  - `javax.slee.management.UnrecognizedSubsystemName-Exception`.
    If the `subsystemName` argument of the method does not identify a internal component or subsystem of the SLEE, as determined by the `getSubsystems` method, an `UnrecognizedSubsystemNameException` is thrown.
  - `javax.slee.InvalidArgumentException`.
    If usage information is requested for a SLEE internal component or subsystem that is not providing usage information, as determined by the `hasUsage` method, an `InvalidArgumentException` is thrown.
  - `javax.slee.management.ManagementException`.
    If the requested operation cannot be performed due to a system-level failure, then the method throws a `ManagementException`. The requested operation may be valid but the SLEE cannot perform the requested operation due to a system-level failure, or the SLEE cannot determine whether the requested operation is valid due to a system-level failure.
  - `java.lang.NullPointerException`.
    If any of a method's arguments are `null`, the method throws a `Null-PointerException`.

14.5.3 `SleeState` objects

Changed in 1.1: String representation and fromString method added.

The `SleeState` class defines an enumerated type for the operational states of the SLEE. A singleton instance of each enumerated value is guaranteed (via an implementation of the `readResolve` method, see `java.io.Serializable` for more details), so that equality tests using == are always evaluated correctly. For convenience, an integer representation of `SleeState` objects is also available.

The following singleton objects define the valid argument or return values for `SleeState` objects.

- `STOPPED`
- `STARTING`
- `RUNNING`
- `STOPPING`

The public contents of the `SleeState` class is as follows:

```
package javax.slee.management;

public final class SleeState implements java.io.Serializable {
       // singletons
       public static final SleeState STOPPED …;
       public static final SleeState STARTING …;
       public static final SleeState RUNNING …;
       public static final SleeState STOPPING …;

       // integer representation
       public static final int SLEE_STOPPED …;
```

```
        public static final int SLEE_STARTING …;
        public static final int SLEE_RUNNING …;
        public static final int SLEE_STOPPING …;

        // string representation
        public static final String STOPPED_STRING = “Stopped”;
        public static final String STARTING_STRING = “Starting”;
        public static final String RUNNING_STRING = “Running”;
        public static final String STOPPING_STRING = “Stopping”;


        // methods
        public boolean isStopped() { … }
        public boolean isStarting(){ … }
        public boolean isRunning() { … }
        public boolean isStopping() { … }

        public static SleeState fromString(String state)
               throws NullPointerException, IllegalArgumentException { … }
        public static SleeState fromInt(int state)
               throws IllegalArgumentException { … }
        public int toInt() { … }
        public boolean equals(Object obj) { … }
        public int hashCode(){ … }
        public String toString(){ … }
}
```

- Each of the `is<State>` methods determines if this `SleeState` object represents the `<State>` operational state, and is equivalent to `(this == <STATE>)`. For example, the `isStopped` method determines if this `SleeState` object represents the Stopped operational state and is equivalent to `(this == STOPPED)`.
- The `fromInt` and `toInt` methods allow conversion between the `SleeState` object form and numeric form.
- The `fromInt` method throws a `java.lang.IllegalArgumentException` if the `state` argument is not one of the four integer state representations.
- The `fromString` method allows conversion between `String` and `SleeState` forms. The conversion is case insensitive. This method throws a `java.lang.NullPointerException` if the argument is `null`. This method throws a `java.lang.IllegalArgumentException` if the argument is not known to this class.

14.5.4 `SleeStateChangeNotification` class

Changed in 1.1: Added support for vendor specific data to be included in the notification.

The public interface of the `SleeStateChangeNotification` class is as follows:

```
package javax.slee.management;

import javax.management.Notification;

public final class SleeStateChangeNotification extends Notification implements VendorEx-
tensions {
        // constructor
        public SleeStateChangeNotification(SleeManagementMBean sleeManagementMBean,
               SleeState newState, SleeState oldState, long sequenceNumber) { … }

        // accessors
        public SleeState getNewState() { … }
        public SleeState getOldState() { … }

        // vendor-specific extension data support
        public static void enableVendorDataSerialization() { … }
        public static void disableVendorDataSerialization() { … }
```

```
        public static void enableVendorDataDeserialization() { … }
        public static void disableVendorDataDeserialization() { … }
        public void setVendorData(Object vendorData) { … }
        public Object getVendorData() { … }

        public String toString() { … }
}
```

The `SleeStateChangeNotification` class inherits all the attributes of the base `Notification` class. Of the inherited attributes, the following attributes have specific meaning defined by the SLEE specification:

- The `type` attribute.
  This attribute specifies the type of the notification being generated. The notification type of all `SleeStateChangeNotification` objects is defined by `SleeManagementMBean.SLEE_STATE_CHANGE_NOTIFICATION_TYPE`.
- The `source` attribute.
  This attribute specifies the `SleeManagementMBean` that emitted this notification.

The `SleeStateChangeNotification` class adds the following additional attributes specific to SLEE state change notifications:

- The `newState` attribute.
  This attribute identifies the state that the SLEE has transitioned to.
- The `oldState` attribute.
  This attribute identifies the previous state the SLEE was in before the state transition.

The following methods are defined by the `SleeStateChangeNotification` class:

- The constructor.
  The constructor takes the following arguments:
  - The `source` argument specifies the value of the inherited `source` attribute for the notification.
  - The `newState` argument specifies the value of the `newState` attribute for the notification.
  - The `oldState` argument specifies the value of the `oldState` attribute for the notification.
  - The `sequenceNumber` argument specifies the value of the inherited `sequenceNumber` attribute for the notification.
- The `getNewState` method.
  This method returns the value of the `newState` attribute.
- The `getOldState` method.
  This method returns the value of the `oldState` attribute.
- The `enableVendorDataSerialization`, `disableVendorDataSerialization`, `enableVendorDataDeserialization`, `disableVendorDataDeserialization`, `setVendorData`, and `getVendorData` methods. *(Added in 1.1)*
  These methods provide support for vendor-specific data in `SleeStateChangeNotification` objects. See Section 14.17 for details and an explanation of these methods.
- The `toString` method.
  This method returns a string representation of the `SleeStateChangeNotification` object.

14.6 `DeploymentMBean` interface

Changed in 1.1: The JMX Object Name of the `DeploymentMBean` *is now specified. An accessor method for library component identifiers has been added. A utility method to obtain the SBB component identifiers of SBBs in a particular Service has also been added.*

The `DeploymentMBean` interface defines the management API for installing and removing components into and out of the SLEE and for interrogating these components.

The Object Name of a `DeploymentMBean` object is defined by the string "`javax.slee.management:name=Deployment`" *(Added in 1.1).*

The Object Name may also be obtained using the `getDeploymentMBean` method on a `SleeProviderMBean` object.

14.6.1 Components that can be installed

Installable components must be packaged into a deployable unit jar file before they can be installed into the SLEE. A deployable unit jar file may contain any number or combination of the following components:

- SBB jar file.
  An SBB jar file contains the class files and deployment descriptors for one or more SBBs (see Section 3.1.9).
- Event jar file.
  An event jar file contains the class files and deployment descriptors for one or more event types (see Section 3.2.3).
- Profile Specification jar file.
  A Profile Specification jar file contains the class files and deployment descriptors for one or more Profile Specifications (see Sections 3.3.9 and 10.4).
- Resource adaptor type jar file.
  A resource adaptor type jar file contains the class files and deployment descriptors for one or more resource adaptor types (see Section 15.3.3).
- Resource adaptor jar file.
  A resource adaptor jar file contains the class files and deployment descriptors for one or more resource adaptors (see Section 15.4.3).
- Service XML file.
  A Service XML file is a deployment descriptor file that contains the definition of one or more Services (see Section 3.4.1).
- Library jar file. *(Added in 1.1)*
  A library jar file contains the class files and deployment descriptor for one or more libraries used by components in the SLEE (see Section 16.3).

A deployable unit jar file may also contain other vendor-specific components.

The deployment descriptor for a deployable unit jar file is described in Section 3.4.5.

14.6.2 Pre-conditions for installing and uninstalling a jar file

A deployable unit cannot be installed successfully in the SLEE under the following conditions:

- A deployable unit with the same URL is already installed in the SLEE. *(Clarified in 1.1)*
- The deployable unit contains a component with the same name, vendor, and version as a component of the same type that is already installed in the SLEE.
- The deployable unit contains components of the same type with the same name, vendor, and version. *(Clarified in 1.1)*

- The deployable unit contains a component that references other components that are not yet installed in the SLEE and are not included in the deployable unit jar. If a component has unresolved dependencies, it cannot be installed. For example, an SBB component may reference event type components and Profile Specification components. If any of the referenced components are not installed or included in the deployable unit jar, then the SBB component has unresolved dependencies and the deployable unit cannot be installed.
- The deployable unit contains components that contain classes, interfaces or deployment descriptors which do not satisfy the requirements and restrictions detailed in this specification.

A deployable unit cannot be uninstalled from the SLEE if there are any dependencies on it from components in other installed deployable units. For example, if a deployable unit contains an SBB jar that depends on a Profile Specification jar contained in a second deployable unit, the deployable unit containing the Profile Specification jar cannot be uninstalled while the deployable unit containing the SBB jar remains installed.

A deployable unit cannot be uninstalled from the SLEE if there are "instances" of components contained in the deployable unit. For example a deployable unit containing a Resource Adaptor cannot be uninstalled if resource adaptor entities of that Resource Adaptor exist. Similarly a deployable unit containing a Profile Specification cannot be uninstalled if Profile Tables exist which have been created from that Profile Specification. *(Clarified in 1.1)*

14.6.3 Deployable unit identifiers and descriptors

Changed in SLEE 1.1: In the SLEE 1.0 specification the deployable unit identifier and the deployable unit descriptor were described by Java interfaces. In the SLEE 1.1 specification these interfaces have been converted to classes. This change does not affect existing management client source code based solely on the SLEE 1.0 interfaces however such code must be recompiled.

When a deployable unit is installed, the SLEE assigns a unique identity to it and returns a deployable unit identifier that encapsulates that identity. The management client uses the deployable unit identifier to query the contents of the deployable unit and eventually uninstall the deployable unit.

A deployable unit is uniquely identified by the URL that it was installed from. *(Added in 1.1)*

The SLEE also creates a deployable unit descriptor for each installed deployable unit. This descriptor encapsulates the relevant deployment information specific to the deployable unit, such as the list of component jars contained within it.

14.6.3.1 `DeployableUnitID` class

The `DeployableUnitID` class encapsulates a deployable unit's identity. A `DeployableUnitID` object is also known as a deployable unit identifier.

A deployable unit is uniquely identified by the URL that it was installed from. *(Added in 1.1)*

The public interface of the `DeployableUnitID` class is as follows:

```
package javax.slee.management;

import java.io.Serializable;

public final class DeployableUnitID implements Comparable, Serializable {
        // constructor
        public DeployableUnitID(String url) { … }

        // identity accessor
        public String getURL() { … }

        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
        public int compareTo(Object obj) { … }
```

```
}
```

- The constructor.
  The constructor takes the following argument:
  - The `url` argument specifies the URL that the deployable unit was installed from.
- The `getURL` method. *(Added in 1.1)*
  This method returns the `url` argument of the constructor.
- The `equals` method.
  This method indicates whether the object specified by the `obj` argument is equal to this deployable unit identifier. The specified object is equal to this deployable unit identifier if the specified object is a deployable unit identifier and it identifies the same deployable unit.
- The `hashCode` method.
  This method provides a hash code value for the deployable unit identifier.
- The `toString` method.
  This method provides a string representation for the deployable unit identifier.
- The `compareTo` method. *(Added in 1.1)*
  This method implements the `java.lang.Comparable` interface allowing deployable unit identifiers to be stored in ordered collections.

14.6.3.2 `DeployableUnitDescriptor` class

The `DeployableUnitDescriptor` class defines the interface of a Java object that describes an installed deployable unit.

The public interface of the `DeployableUnitDescriptor` class is as follows:

```
package javax.slee.management;

import java.io.Serializable;
import java.util.Date;
import javax.slee.ComponentID;

public class DeployableUnitDescriptor implements VendorExtensions, Serializable {
        // constructor
        public DeployableUnitDescriptor(DeployableUnitID id, Date deploymentDate,
                ComponentID[] components) { … }

        // accessors
        public DeployableUnitID getID() { … }
        public String getURL() { … }
        public Date getDeploymentDate() { … }
        public ComponentID[] getComponents() { … }

        // vendor-specific extension data support
        public static void enableVendorDataSerialization() { … }
        public static void disableVendorDataSerialization() { … }
        public static void enableVendorDataDeserialization() { … }
        public static void disableVendorDataDeserialization() { … }
        public void setVendorData(Object vendorData) { … }
        public Object getVendorData() { … }

        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:

  - The `id` argument specifies the `DeployableUnitID` identifier that identifies the deployable unit described by this `DeployableUnitDescriptor`.
  - The `deploymentDate` argument specifies the date that the deployable unit was installed into the SLEE.
  - The `components` argument specifies an array of `ComponentID` objects that identify the SLEE components that are contained in the deployable unit.

- The `getID` method. *(Added in 1.1)*
  This method returns the `id` argument of the constructor.
- The `getURL` method.
  This method returns the URL that the deployable unit was installed from.
- The `getDeploymentDate` method.
  This method returns the `deploymentDate` argument of the constructor.
- The `getComponents` method.
  This method returns the `components` argument of the constructor.
- The `enableVendorDataSerialization`, `disableVendorDataSerialization`, `enableVendorDataDeserialization`, `disableVendorDataDeserialization`, `setVendorData`, and `getVendorData` methods. *(Added in 1.1)*
  These methods provide support for vendor-specific data in `DeployableUnitDescriptor` objects. See Section 14.17 for details and an explanation of these methods.
- The `equals` method.
  This method indicates whether the object specified by the `obj` argument is equal to this deployable unit descriptor. The specified object is equal to this deployable unit descriptor if the specified object is a deployable unit descriptor and contains a `DeployableUnitID` that identifies the same deployable unit as this deployable unit descriptor.
- The `hashCode` method.
  This method provides a hash code value for the deployable unit descriptor.
- The `toString` method.
  This method provides a string representation for the deployable unit descriptor.

14.6.4 Component identity

Changed in 1.1: In the SLEE 1.0 specification component identifiers and component descriptors were described by Java interfaces. In the SLEE 1.1 specification these interfaces have been converted to classes. This change does not affect existing management client source code based solely on the SLEE 1.0 interfaces however such source code must be recompiled.

Each component that is installed into the SLEE is assigned a unique identity. The SLEE establishes component identity from information contained in the component's deployment descriptor. Component identity is immutable while the component is installed in the SLEE. Two components of the same type that have the same component identity cannot be installed simultaneously in the SLEE.

- SBB component identity.
  An SBB's name, vendor, and version determine the SBB's component identity.
- Event type component identity.
  An event type's name, vendor, and version determine the event type's component identity.
- Profile Specification component identity.
  A Profile Specification's name, vendor, and version determine the Profile Specification's component identity.

- Service component identity.
  A Service's name, vendor, and version determine the Service's component identity.
- Resource adaptor type component identity.
  A resource adaptor type's name, vendor, and version determine the resource adaptor type's component identity.
- Resource adaptor component identity.
  A resource adaptor's name, vendor, and version determine the resource adaptor's component identity.
- Library component identity. *(Added in 1.1)*
  A library's name, vendor, and version determine the library's component identity.

14.6.5 Component identifiers

Changed in 1.1: Component identity interfaces changed to classes.

A component identifier is a Java object that encapsulates a component's identity. The SLEE specification defines the following component identifier classes.

- The `ComponentID` class is the common base class of all component identifier classes.
- The `SbbID` class for component identifiers that encapsulate SBB component identities.
- The `EventTypeID` class for component identifiers that encapsulate event type component identities.
- The `ProfileSpecificationID` class for component identifiers that encapsulate Profile Specification component identities.
- The `ServiceID` class for component identifiers that encapsulate Service component identities.
- The `ResourceAdaptorTypeID` class for component identifiers that encapsulate resource adaptor type component identities.
- The `ResourceAdaptorID` class for component identifiers that encapsulate resource adaptor component identities.
- The `LibraryID` class for component identifiers that encapsulate library component identifiers. *(Added in 1.1)*

14.6.5.1 `ComponentID` class

The `ComponentID` class is the common base class for all component identifier interfaces. All Java objects that extend the `ComponentID` class are also known as component identifiers.

The public interface of the `ComponentID` class is as follows:

```
package javax.slee;

import java.io.Serializable;

public abstract class ComponentID implements Comparable, Serializable {
        // constructor
        protected ComponentID(String name, String vendor, String version) { … }

        // identity accessors
        public final String getName() { … }
        public final String getVendor() { … }
        publib final String getVersion() { … }

        public final boolean equals(Object obj) { … }
        public final int hashCode() { … }
        public final String toString() { … }
```

```
        public final int compareTo(Object obj) { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `name`, `vendor`, and `version` arguments specifies the name, vendor, and version respectively of the component identified by this component identifier.
- The `getName`, `getVendor`, and `getVersion` methods. (Added in 1.1)
  These methods return the `name`, `vendor`, and `version` arguments respectively of the constructor.
- The `equals` method.
  This method indicates whether the object specified by the `obj` argument is equal to this component identifier. The specified object is equal to this component identifier if the specified object is a component identifier of the same type (as determined by the concrete subclass of the component identifier) and has the same identity.
- The `hashCode` method.
  This method provides a hash code value for the component identifier.
- The `toString` method.
  This method provides a string representation for the component identifier.
- The `compareTo` method. *(Added in 1.1)*
  This method implements the java.lang.`Comparable` interface allowing component identifiers to be stored in ordered collections.

14.6.5.2 `SbbID` class

The `SbbID` class encapsulates an SBB's component identity. An `SbbID` object is also known as an SBB identifier.

The public interface of the `SbbID` class is as follows:

```
package javax.slee;

public final class SbbID extends ComponentID {
        // constructor
        public SbbID(String name, String vendor, String version) { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `name`, `vendor`, and `version` arguments specifies the name, vendor, and version respectively of the SBB identified by this SBB identifier.

14.6.5.3 `EventTypeID` class

The `EventTypeID` class encapsulates an event type's component identity. An `EventTypeID` object is also known as an event type identifier.

The public interface of the `EventTypeID` class is as follows:

```
package javax.slee;

public final class EventTypeID extends ComponentID {
        // constructor
        public EventTypeID(String name, String vendor, String version) { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `name`, `vendor`, and `version` arguments specifies the name, vendor, and version respectively of the event type identified by this event type component identifier.

14.6.5.4 `ProfileSpecificationID` class

The `ProfileSpecificationID` class encapsulates a Profile Specification's component identity. A `ProfileSpecificationID` object is also known as a Profile Specification identifier.

The public interface of the `ProfileSpecificationID` class is as follows:

```
package javax.slee.profile;

public final class ProfileSpecificationID extends ComponentID {
       // constructor
       public ProfileSpecificationID(String name, String vendor, String version) { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `name`, `vendor`, and `version` arguments specifies the name, vendor, and version respectively of the profile specification identified by this profile specification identifier.

14.6.5.5 `ServiceID` class

The `ServiceID` class encapsulates a Service's component identity. A `ServiceID` object is also known as a Service identifier.

The public interface of the `ServiceID` interface is as follows:

```
package javax.slee;

public final class ServiceID extends ComponentID {
       // constructor
       public ServiceID(String name, String vendor, String version) { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `name`, `vendor`, and `version` arguments specifies the name, vendor, and version respectively of the Service identified by this Service identifier.

14.6.5.6 `ResourceAdaptorTypeID` class

The `ResourceAdaptorTypeID` class encapsulates a resource adaptor type's (see Section 15.3) component identity. A `ResourceAdaptorTypeID` object is also known as a resource adaptor type identifier.

The public interface of the `ResourceAdaptorTypeID` class is as follows:

```
package javax.slee.resource;

public final class ResourceAdaptorTypeID extends ComponentID {
       // constructor
       public ResourceAdaptorTypeID(String name, String vendor, String version) { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:

- The `name`, `vendor`, and `version` arguments specifies the name, vendor, and version respectively of the resource adaptor type identified by this resource adaptor type identifier.

14.6.5.7 `ResourceAdaptorID` class

The `ResourceAdaptorID` class encapsulates a resource adaptor's (see Section 15.4) component identity. A `ResourceAdaptorID` object is also known as a resource adaptor identifier.

The public interface of the `ResourceAdaptorID` class is as follows:

```
package javax.slee.resource;

public final class ResourceAdaptorID extends ComponentID {
        // constructor
        public ResourceAdaptorID(String name, String vendor, String version) { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `name`, `vendor`, and `version` arguments specifies the name, vendor, and version respectively of the resource adaptor identified by this resource adaptor identifier.

14.6.5.8 `LibraryID` class

Added in 1.1.

The `LibraryID` class encapsulates an library's component identity. A `LibraryID` object is also known as a library identifier.

The public interface of the `LibraryID` class is as follows:

```
package javax.slee.management;

import javax.slee.ComponentID;

public final class LibraryID extends ComponentID {
        // constructor
        public LibraryID(String name, String vendor, String version) { … }
}
```

- The constructor.
  The constructor takes the following arguments:
  - The `name`, `vendor`, and `version` arguments specifies the name, vendor, and version respectively of the library identified by this library identifier.

14.6.6 Common characteristics of component identifier classes

The following are the common characteristics for the deployable unit and component identifier classes:

- A deployable unit identifier object or component identifier object is immutable. The values returned from the methods in a deployable unit identifier object or component identifier object do not change over the lifetime of the object.

14.6.7 Component descriptors

For each installed component, the SLEE also creates a component descriptor. A component descriptor encapsulates the relevant deployment information specific to the installed component.

- A management client that has a component identifier for an installed component can obtain a corresponding component descriptor by querying a `DeploymentMBean` object.

14.6.7.1 `ComponentDescriptor` class

The `ComponentDescriptor` class defines the common operations that are relevant to all component descriptors. All specific component descriptor classes extend from this common base class.

The public interface of the `ComponentDescriptor` class is follows:

```
package javax.slee.management;

import java.io.Serializable;
import javax.slee.ComponentID;

public abstract class ComponentDescriptor implements VendorExtensions, Serializable {
        // constructor
        protected ComponentDescriptor(ComponentID component,
                DeployableUnitID deployableUnit, String source,
                LibraryID[] libraries) { … }

        // accessors
        public final ComponentID getID() { … }
        public final String getSource() { … }
        public final DeployableUnitID getDeployableUnit() { … }
        public final String getName() { … }
        public final String getVendor() { … }
        public final String getVersion() { … }
        public final LibraryID[] getLibraries() { … }

        // vendor-specific extension data support
        public static void enableVendorDataSerialization() { … }
        public static void disableVendorDataSerialization() { … }
        public static void enableVendorDataDeserialization() { … }
        public static void disableVendorDataDeserialization() { … }
        public void setVendorData(Object vendorData) { … }
        public Object getVendorData() { … }

        public final boolean equals(Object obj) { … }
        public final int hashCode() { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `component` argument specifies the component identifier for the component being described.
  - The `deployableUnit` argument specifies the deployable unit identifier that identifies the deployable unit this component came from.
  - The `source` argument specifies the name of the component jar file in the deployable unit from which the component was installed, or in the case of Service components, the name of the Service XML file from which the Service was installed. This name is identical to the name provided in the relevant deployment descriptor element in the deployable unit jar file from where the component was installed.
  - The `libraries` argument specifies an array of `LibraryID` objects that identify the library components that are used by this component.
- The `getID` method.
  This method returns the `component` argument of the constructor.
- The `getSource` method.
  This method returns the `source` argument of the constructor.
- The `getDeployableUnit` method.
  This method returns the `deployableUnit` argument of the constructor.

- The `getName` method.
  This method returns the name of the component.
- The `getVendor` method.
  This method returns the vendor of the component.
- The `getVersion` method.
  This method returns the version of the component.
- The `getLibraries` method. *(Added in 1.1)*
  This method returns the `libraries` argument of the constructor.
- The `enableVendorDataSerialization`, `disableVendorDataSerialization`, `enableVendorDataDeserialization`, `disableVendorDataDeserialization`, `setVendorData`, and `getVendorData` methods. *(Added in 1.1)*
  These methods provide support for vendor-specific data in `ComponentDescriptor` objects. See Section 14.17 for details and an explanation of these methods.
- The `equals` method. *(Added in 1.1)*
  This method indicates whether the object specified by the `obj` argument is equal to this component descriptor. The specified object is equal to this component descriptor if the specified object is a component descriptor and contains a ComponentID that identifies the same component as this component descriptor.
- The `hashCode` method. *(Added in 1.1)*
  This method provides a hash code value for the component descriptor.

14.6.7.2 `SbbDescriptor` class

The `SbbDescriptor` class defines the attributes that describe an SBB that is installed in the SLEE.

The public interface of the `SbbDescriptor` class is as follows:

```
package javax.slee.management;

import javax.slee.EventTypeID;
import javax.slee.SbbID;
import javax.slee.profile.ProfileSpecificationID;
import javax.slee.resource.ResourceAdaptorTypeID;

public class SbbDescriptor extends ComponentDescriptor {
        // constructor
        public SbbDescriptor(SbbID component, DeployableUnitID deployableUnit,
                String source, LibraryID[] libraries, SbbID[] sbbs,
                EventTypeID[] eventTypes, ProfileSpecificationID[] profileSpecs,
                ProfileSpecificationID addressProfileSpec,
                ResourceAdaptorTypeID[] raTypes, String[] linkNames) { … }

        // accessors
        public final SbbID[] getSbbs() { … }
        public final EventTypeID[] getEventTypes() { … }
        public final ProfileSpecificationID[] getProfileSpecifications() { … }
        public final ProfileSpecificationID getAddressProfileSpecification() { … }
        public final ResourceAdaptorTypeID[] getResourceAdaptorTypes() { … }
        public final String[] getResourceAdaptorEntityLinks() { … }

        public String toString() { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `component` argument specifies the component identifier for the SBB being described.

  - The `deployableUnit` argument specifies the deployable unit identifier that identifies the deployable unit this SBB came from.
  - The `source` argument specifies the name of the component jar file in the deployable unit from which the SBB was installed.
  - The `libraries` argument specifies an array of `LibraryID` objects that identify the library components that are used by the described SBB (through the `library-ref` elements in the SBB's deployment descriptor).
  - The `sbbs` argument specifies an array of SbbID object that identify the SBBs used by the described SBB (through the `sbb-ref` elements in the SBB's deployment descriptor).
  - The `eventTypes` argument specifies an array of `EventTypeID` objects that identify the event types used by the described SBB (through the `event` elements in the SBB's deployment descriptor).
  - The `profileSpecs` argument specifies an array of `ProfileSpecificationID` objects that identify the Profile Specifications used by the described SBB (through the `profile-spec-ref` elements in the SBB's deployment descriptor).
  - The `addressProfileSpec` argument specifies a `ProfileSpecificationID` object that identifies the Address Profile Specification of the described SBB (through the `address-profile-spec-alias-ref` element in the SBB's deployment descriptor)
  - The `raTypes` argument specifies an array of `ResourceAdaptorTypeID` objects that identify the resource adaptor types referenced by the described SBB (through the `resource-adaptor-type-binding` elements in the SBB's deployment descriptor).
  - The `linkNames` argument specifies an array of `String` objects which name the resource adaptor entity links referenced by the described SBB (through the `resource-adaptor-entity-link` elements in the SBB's deployment descriptor).

- The `getSbbs` method.
  This method returns the `sbbs` argument of the constructor.
- The `getEventTypes` method.
  This method returns the `eventTypes` argument of the constructor.
- The `getProfileSpecifications` method.
  This method returns the `profileSpecs` argument of the consructor.
- The `getAddressProfileSpecification` method.
  This method returns the `addressProfileSpec` argument of the constructor.
- The `getResourceAdaptorTypes` method.
  This method returns the `raTypes` argument of the constructor.
- The `getResourceAdaptorEntityLinks` method.
  This method returns the `linkNames` argument of the constructor.
- The `toString` method. *(Added in 1.1)*
  This method provides a string representation for the SBB component descriptor.

14.6.7.3 `EventTypeDescriptor` class

The `EventTypeDescriptor` class defines the attributes that describe an event type that is installed in the SLEE.

The public interface of the `EventTypeDescriptor` class is as follows:

```
package javax.slee.management;

import javax.slee.EventTypeID;

public class EventTypeDescriptor extends ComponentDescriptor {
        // constructor
        public EventTypeDescriptor(EventTypeID component, DeployableUnitID deployableUnit,
                String source, LibraryID[] libraries, String eventClass) { … }

        // accessors
        public final String getEventClassName() { … }

        public String toString() { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `component` argument specifies the component identifier for the event type being described.
  - The `deployableUnit` argument specifies the deployable unit identifier that identifies the deployable unit this event type came from.
  - The `source` argument specifies the name of the component jar file in the deployable unit from which the event type was installed.
  - The `libraries` argument specifies an array of `LibraryID` objects that identify the library components that are used by the described event type (through the `library-ref` elements in the event type jar deployment descriptor).
  - The `eventClass` argument specifies the name of the interface or class used as the event class for event type.
- The `getEventClassName` method. *(Added in 1.1)*
  This method returns the `eventClass` argument of the constructor.
- The `toString` method. *(Added in 1.1)*
  This method provides a string representation for the event type component descriptor.

14.6.7.4 `ProfileSpecificationDescriptor` class

The `ProfileSpecificationDescriptor` class defines the attributes that describe a Profile Specification that is installed in the SLEE.

The public interface of the `ProfileSpecificationDescriptor` class is as follows:

```
package javax.slee.profile;

import javax.slee.management.ComponentDescriptor;
import javax.slee.management.DeployableUnitID;
import javax.slee.management.LibraryID;

public class ProfileSpecificationDescriptor extends ComponentDescriptor {
        // constructor
        public ProfileSpecificationDescriptor(ProfileSpecificationID component,
                DeployableUnitID deployableUnit, String source, LibraryID[] libraries,
                ProfileSpecificationID[] profileSpecs, String cmpInterface,
                String[] indexedAttrs) { … }

        // accessors
        public final ProfileSpecificationID[] getProfileSpecifications() { … }
        public final String getCMPInterfaceName() { … }
        public final String[] getIndexedAttributes() { … }
```

```
        public String toString() { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `component` argument specifies the component identifier for the Profile Specification being described.
  - The `deployableUnit` argument specifies the deployable unit identifier that identifies the deployable unit this Profile Specification came from.
  - The `source` argument specifies the name of the component jar file in the deployable unit from which the Profile Specification was installed.
  - The `libraries` argument specifies an array of `LibraryID` objects that identify the library components that are used by the described Profile Specification (through the `library-ref` elements in the Profile Specification's deployment descriptor).
  - The `profileSpecs` argument specifies an array of `ProfileSpecificationID` objects that identify the Profile Specifications used by the described Profile Specification (through the `profile-spec-ref` elements in the Profile Specification's deployment descriptor).
  - The `cmpInterface` argument specifies the name of the Profile CMP interface of the Profile Specification.
  - The `indexedAttrs` argument specifies the names of the attributes of the Profile Specification which have been indexed. Only indexed attributes may be used in query expressions.
- The `getProfileSpecifications` method. *(Added in 1.1)*
  This method returns the `profileSpecs` argument of the consructor.
- The `getCMPInterfaceName` method.
  This method returns the `cmpInterface` argument of the constructor.
- The `getIndexedAttributes` method. *(Added in 1.1)*
  This method returns the `indexedAttrs` argument of the constructor.
- The `toString` method. *(Added in 1.1)*
  This method provides a string representation for the Profile Specification component descriptor.

14.6.7.5 `ServiceDescriptor` class

The `ServiceDescriptor` class defines the attributes that describe a Service that is installed in the SLEE.

The public interface of the `ServiceDescriptor` class is as follows:

```
package javax.slee.management;

import javax.slee.SbbID;
import javax.slee.ServiceID;

public class ServiceDescriptor extends ComponentDescriptor {
        // constructor
        public ServiceDescriptor(ServiceID component, DeployableUnitID deployableUnit,
                String source, SbbID rootSbb, String addressProfileTable,
                String resourceInfoProfileTable) { … }

        // accessors
        public final SbbID getRootSbb() { … }
        public final String getAddressProfileTable() { … }
```

```
        // deprecated
        public final String getResourceInfoProfileTable() { … }

        public String toString() { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `component` argument specifies the component identifier for the Service being described.
  - The `deployableUnit` argument specifies the deployable unit identifier that identifies the deployable unit this Service came from.
  - The `source` argument specifies the name of the Service XML file in the deployable unit from which the Service was installed.
  - The `rootSbb` argument specifies the SBB identifier object that identifies the root SBB component of the Service.
  - The `addressProfileTable` argument specifies the name of the Address Profile Table of the Service. If the Service does not have an Address Profile Table, this argument should be `null`.
  - The `resourceInfoProfileTable` argument specifies the name of the Resource Info Profile Table of the Service. If the Service does not have a Resource Info Profile Table, this argument should be `null`.
    (*Deprecated in 1.1: The Resource Info Profile Table has been deprecated. The Resource Adaptor architecture defined by the SLEE 1.1 specification describes how Resource Adaptors may interact with profile tables and profiles. The ability to specify a Resource Info Profile Table for a Service may be removed in a future version of this specification.)*
- The `getRootSbb` method.
  This method returns the `rootSbb` argument of the constructor.
- The `getAddressProfileTable` method.
  This method returns the `addressProfileTable` argument of the constructor.
- The `getResourceInfoProfileTable` method. *(Deprecated in SLEE 1.1)*
  This method returns the `resourceInfoProfileTable` argument of the constructor.
- The `toString` method. *(Added in 1.1)*
  This method provides a string representation for the Service component descriptor.

14.6.7.6 `ResourceAdaptorTypeDescriptor` class

The `ResourceAdaptorTypeDescriptor` class defines the attributes that describe a resource adaptor type that is installed in the SLEE.

The public interface of the `ResourceAdaptorTypeDescriptor` class is as follows:

```
package javax.slee.resource;

import javax.slee.EventTypeID;
import javax.slee.management.ComponentDescriptor;
import javax.slee.management.DeployableUnitID;
import javax.slee.management.LibraryID;

public class ResourceAdaptorTypeDescriptor extends ComponentDescriptor {
        // constructor
```

```
        public ResourceAdaptorTypeDescriptor(ResourceAdaptorTypeID component,
               DeployableUnitID deployableUnit, String source, LibraryID[] libraries,
               String[] activityTypes, String raInterface,
               EventTypeID[] eventTypes) { … }

        // accessors
        public final String[] getActivityTypes() { … }
        public final String getResourceAdaptorInterface() { … }
        public final EventTypeID[] getEventTypes() { … }

        public String toString() { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `component` argument specifies the component identifier for the resource adaptor type being described.
  - The `deployableUnit` argument specifies the deployable unit identifier that identifies the deployable unit this resource adaptor type came from.
  - The `source` argument specifies the name of the component jar file in the deployable unit from which the resource adaptor type was installed.
  - The `libraries` argument specifies an array of `LibraryID` objects that identify the library components that are used by the described resource adaptor type (through the `library-ref` elements in the resource adaptor type's deployment descriptor).
  - The `activityTypes` specifies an array of String objects that identify the names of the interfaces and/or classes of the activity types used by the described resource adaptor type.
  - The `raInterface` argument specifies the name of the resource adaptor interface exposed by the resource adaptor type. If the resource adaptor type does not have a resource adaptor interface, this argument should be `null`.
  - The `eventTypes` argument specifies an array of `EventTypeID` objects that identify the event types used by the described resource adaptor type.
- The `getActivityTypes` method. *(Added in 1.1)*
  This method returns the `activityTypes` argument of the constructor.
- The `getResourceAdaptorInterface` method. *(Added in 1.1)*
  This method returns the `raInterface` argument of the constructor.
- The `getEventTypes` method.
  This method returns the `eventTypes` argument of the constructor.
- The `toString` method. *(Added in 1.1)*
  This method provides a string representation for the resource adaptor type component descriptor.

14.6.7.7 `ResourceAdaptorDescriptor` class

The `ResourceAdaptorDescriptor` class defines the attributes that describe a resource adaptor that is installed in the SLEE.

The public interface of the `ResourceAdaptorDescriptor` class is as follows:

```
package javax.slee.resource;

import javax.slee.profile.ProfileSpecificationID;

import javax.slee.management.ComponentDescriptor;
import javax.slee.management.DeployableUnitID;
import javax.slee.management.LibraryID;
```

```
public class ResourceAdaptorDescriptor extends ComponentDescriptor {
        // constructor
        public ResourceAdaptorDescriptor(ResourceAdaptorID component,
                DeployableUnitID deployableUnit, String source, LibraryID[] libraries,
                ResourceAdaptorTypeID[] raTypes, ProfileSpecificationID[] profileSpecs,
                boolean supportsActiveReconfiguration) { … }

        // accessors
        public final ResourceAdaptorTypeID[] getResourceAdaptorTypes() { … }
        public final ProfileSpecificationID[] getProfileSpecifications() { … }
        public final boolean supportsActiveReconfiguration() { … }

        // deprecated
        public final ResourceAdaptorTypeID getResourceAdaptorType() { … }

        public String toString() { … }
}
```

- The constructor. *(Added in 1.1)*
  The constructor takes the following arguments:
  - The `component` argument specifies the component identifier for the resource adaptor being described.
  - The `deployableUnit` argument specifies the deployable unit identifier that identifies the deployable unit this resource adaptor came from.
  - The `source` argument specifies the name of the component jar file in the deployable unit from which the resource adaptor was installed.
  - The `libraries` argument specifies an array of `LibraryID` objects that identify the library components that are used by the described resource adaptor (through the `library-ref` elements in the resource adaptor's deployment descriptor).
  - The `raTypes` argument specifies an array of `ResourceAdaptorTypeID` objects that identify the resource adaptor types referenced by the described resource adaptor (through the `resource-adaptor-type-ref` elements in the resource adaptor's deployment descriptor).
  - The `profileSpecs` argument specifies an array of `ProfileSpecificationID` objects that identify the Profile Specifications used by the described resource adaptor (through the `profile-spec-ref` elements in the resource adaptor's deployment descriptor).
  - The `supportsActiveReconfiguration` argument specifies a `boolean` value equivalent to the value of the `supports-active-reconfiguration` attribute of the `resource-adaptor-class` element in the described resource adaptor's deployment descriptor
- The `getResourceAdaptorTypes` method. *(Added in 1.1)*
  This method returns the `raTypes` argument of the constructor.
- The `getProfileSpecifications` method. *(Added in 1.1)*
  This method returns the `profileSpecs` argument of the constructor.
- The `supportsActiveReconfiguration` method. *(Added in 1.1)*
  This method returns the `supportsActiveReconfiguration` argument of the constructor.
- The `getResourceAdaptorType` method.
  Deprecated in 1.1: The SLEE 1.1 specification allows Resource Adaptors to implement more than one resource adaptor type, therefore this method has been deprecated.

This method will return the first array index of the array returned by the `getResource-AdaptorTypes` method, which replaces this method.

- The `toString` method. *(Added in 1.1)*
  This method provides a string representation for the resource adaptor component descriptor.

14.6.7.8 `LibraryDescriptor` class

Added in 1.1.

The `LibraryDescriptor` class defines the attributes that describe a library that is installed in the SLEE.

The public interface of the `LibraryDescriptor` class is as follows:

```
package javax.slee.management;

public class LibraryDescriptor extends ComponentDescriptor {
        // constructor
        public LibraryDescriptor(LibraryID component, DeployableUnitID deployableUnit,
                String source, LibraryID[] libraries, String[] libraryJars) { … }

        // accessors
        public final String[] getLibraryJars() { … }

        public String toString() { … }
}
```

- The constructor.
  The constructor takes the following arguments:
  - The `component` argument specifies the component identifier for the library being described.
  - The `deployableUnit` argument specifies the deployable unit identifier that identifies the deployable unit this library came from.
  - The `source` argument specifies the name of the component jar file in the deployable unit from which the library was installed.
  - The `libraries` argument specifies an array of `LibraryID` objects that identify the library components that are used by the described library (through the `library-ref` elements in the library's deployment descriptor).
  - The `libraryJars` argument specifies an array of `String` objects that identify the names of the constituent jars included in the library. This name is identical to the name provided in the `jar` element in the library's deployment descriptor.
- The `getLibraryJars` method.
  This method returns the `libraryJars` argument of the constructor.
- The `toString` method.
  This method provides a string representation for the library component descriptor.

14.6.8 `DeploymentMBean` interface

The `DeploymentMBean` interface is as follows:

```
package javax.slee.management;

import java.net.MalformedURLException
import javax.slee.EventTypeID;
import javax.slee.InvalidStateException;
import javax.slee.SbbID;
import javax.slee.ServiceID;
```

```
import javax.slee.UnrecognizedComponentException;
import javax.slee.UnrecognizedServiceException;
import javax.slee.profile.ProfileSpecificationID;
import javax.slee.resource.ResourceAdaptorID;
import javax.slee.resource.ResourceAdaptorTypeID;

public interface DeploymentMBean {
        // JMX Object Name string of the DeploymentMBean
        public static final String OBJECT_NAME = "javax.slee.management:name=Deployment";

        // methods
        public DeployableUnitID install(String url)
                throws  NullPointerException,
                        MalformedURLException,
                        AlreadyDeployedException,
                        DeploymentException,
                        ManagementException;
        public void uninstall(DeployableUnitID id)
                throws  NullPointerException,
                        UnrecognizedDeployableUnitException,
                        DependencyException,
                        InvalidStateException,
                        ManagementException;
        public DeployableUnitID getDeployableUnit(String url)
                throws  NullPointerException,
                        UnrecognizedDeployableUnitException,
                        ManagementException;
        public DeployableUnitID[] getDeployableUnits()
                throws  ManagementException;
        public SbbID[] getSbbs()
                throws ManagementException;
        public SbbID[] getSbbs(ServiceID service)
                throws  NullPointerException,
                        UnrecognizedServiceException,
                        ManagementException;
        public EventTypeID[] getEventTypes()
                throws  ManagementException;
        public ProfileSpecificationID[] getProfileSpecifications()
                throws  ManagementException;
        public ServiceID[] getServices()
                throws  ManagementException;
        public ResourceAdaptorTypeID[] getResourceAdaptorTypes()
                throws  ManagementException;
        public ResourceAdaptorID[] getResourceAdaptors()
                throws  ManagementException;
        public LibraryID[] getLibraries()
                throws  ManagementException;
        public ComponentID[] getReferringComponents(ComponentID id)
                throws  NullPointerException,
                        UnrecognizedComponentException,
                        ManagementException;
        public DeployableUnitDescriptor getDescriptor(DeployableUnitID id)
                throws  NullPointerException,
                        UnrecognizedDeployableUnitException,
                        ManagementException;
        public DeployableUnitDescriptor[] getDescriptors(DeployableID[] ids)
                throws  NullPointerException,
                        ManagementException;
        public boolean isInstalled(DeployableUnitID id)
                throws  NullPointerException,
                        ManagementException;
        public ComponentDescriptor getDescriptor(ComponentID id)
                throws  NullPointerException,
                        UnrecognizedComponentException,
                        ManagementException;
        public ComponentDescriptor[] getDescriptors(ComponentID[] ids)
                throws  NullPointerException,
                        ManagementException;
```

```
        public boolean isInstalled(ComponentID id)
               throws  NullPointerException,
                       ManagementException;
}
```

- The `OBJECT_NAME` constant. *(Added in 1.1)*
  This constant defines the string form of the `DeploymentMBean`'s JMX Object Name.
- The `install` method. *(Changed in 1.1: to take library components into consideration)*
  This method installs the deployable unit jar file identified by the `url` argument. Each component jar file contained in the deployable unit is installed by the SLEE. If all these components install successfully, the SLEE then installs any Services contained in the deployable unit jar file. The deployable unit is only successfully installed if all its included component jars and Services install successfully. If an installation attempt fails, any components in the deployable unit that were successfully installed must be uninstalled again before the `install` method returns.
  A component cannot be installed if it has the same identity as a component of the same type already installed in the SLEE. Additionally, the SLEE must make other considerations depending on the type of the component being installed.
  - An SBB jar file.
    An SBB contained in an SBB jar file may have references to other SBBs, event types, Profile Specifications, resource adaptor types or libraries. These other components must be installed before the SBB jar file can be successfully installed.
  - An event jar file.
    An event jar file contains one or more event types. An event jar file cannot be installed successfully if one or more of the event types defined in the event jar file have the same identity as an event already known to the SLEE, regardless who defined the existing event type or the new event type. An SBB developer, a resource, a resource adaptor type, a SLEE vendor, or the SLEE specification may have defined the existing event type. If an event jar references one or more libraries, these library components must be installed before the event jar can be successfully installed.
  - A Profile Specification jar file.
    A Profile Specification in a Profile Specification jar may include a reference to one or more libraries. If a Profile Specification in a Profile Specification jar includes such references, the referenced library components must be installed before the Profile Specification jar can be successfully installed.
  - A Service XML file.
    A Service described in a Service XML file can only be installed if its root SBB is already installed.
  - A resource adaptor type jar file.
    A resource adaptor type jar file contains one or more resource adaptor types. A resource adaptor type may have references to event types and libraries. These event types and/or libraries must be installed before the resource adaptor type jar file can be successfully installed.
  - A resource adaptor jar file.
    A resource adaptor in a resource adaptor jar file contains references to one or more resource adaptor types, and optionally to one or more libraries. A resource adaptor jar file can only be installed if the resource adaptor types and libraries referenced by the resource adaptors in the resource adaptor jar file are already installed.
  - A library jar file. *(Added in 1.1)*
    A library in a library jar may contain references to other libraries. A library jar can only be installed successfully if the referenced libraries have already been installed (unless the referenced libraries are also included in the library jar).

The `install` method may throw the following exceptions.

- `java.net.MalformedURLException`. *(Clarified in 1.1)*
  This exception is thrown if the `url` parameter is not a properly formatted URL.
- `javax.slee.management.AlreadyDeployedException`.
  If a deployable unit with the same URL is already installed, or the deployable unit contains a component with the same component identity as a component of the same type that is already installed in the SLEE, an `AlreadyDeployedException` exception is thrown.
- `javax.slee.management.DeploymentException`.
  This exception is thrown if a component could not be installed due to a problem directly related to it or its contents. For example a contained component may fail a SLEE compliance check, a referenced component may not be installed, or a required class file might be missing from the jar.

- The `uninstall` method. *(Changed in 1.1 to include library component deployable units)*
  This method uninstalls the deployable unit specified by the `id` argument. Uninstalling a deployable unit causes all the components in the deployable unit to be uninstalled. A deployable unit cannot be uninstalled if any component contained in any other deployable unit currently installed in the SLEE depends on a component contained in the deployable unit being uninstalled.
  - An SBB jar file cannot be uninstalled while the SBB defined in the SBB jar file is referenced by other SBBs or Services installed in the SLEE. The `getReferringComponents` method may be used to determine the set of SBBs and Services that reference an SBB.
  - An event jar file cannot be uninstalled while any of the event types defined in the event jar file are referenced by SBBs or resource adaptor types installed in the SLEE. The `getReferringComponents` method may be used to determine the set of SBBs and resource adaptor types that reference an event type.
  - A Profile Specification jar file cannot be uninstalled while SBBs or resource adaptors that depend on it are installed. The `getReferringComponents` method may be used to determine the set of SBBs and resource adaptors that reference a Profile Specification.
  - No special considerations need to be made by the SLEE when uninstalling a Service XML file.
  - A resource adaptor type jar file cannot be uninstalled while SBBs or resource adaptors that depend on it are installed. The `getReferringComponents` method may be used to determine the set of SBBs and resource adaptors that reference a resource adaptor type.
  - A resource adaptor jar file cannot be uninstalled while resource adaptor entities exist which have been created from resource adaptors defined in the resource adaptor jar.
  - A library jar file cannot be uninstalled while a component from any other deployable unit that depends on a library included in the library jar file remains installed in the SLEE. *(Added in 1.1)*

  This method may throw the following exceptions.
  - `javax.slee.management.UnrecognizedDeployableUnitException`.
    This exception is thrown if the deployable unit identifier passed in as an argument is not recognized by the SLEE as a valid identifier.
  - `javax.slee.management.DependencyException`.
    This exception is thrown if one or more of the components included in the deployable unit are being referenced by a component in another installed deployable unit.

  - `javax.slee.InvalidStateException.`
    This exception is thrown if one or more of the components included in the deployable unit cannot be uninstalled because the component is in a state that does not allow the component to be uninstalled. For example, a Service cannot be uninstalled while it is in the "Active" state.

- The `getDeployableUnit` method.
  This method returns a `DeployableUnitID` object when provided with the URL of a deployable unit installed in the SLEE.

  This method may throw the following exception.

  - `javax.slee.management.UnrecognizedDeployableUnitException`.
    This exception is thrown if the URL passed as an argument does not correspond to a deployable unit installed in the SLEE.

- The `getDeployableUnits` method.
  This method returns an array of `DeployableUnitID` objects containing the identifiers of all the deployable units installed in the SLEE.

- The `getSbbs()` method.
  This method returns an array of `SbbID` objects containing the identifiers of all the SBBs installed in the SLEE.

- The `getSbbs(ServiceID service)` method. *(Added in 1.1)*
  This method returns an array of `SbbID` objects containing only the identifiers of the SBBs that participate in the service specified by the `service` argument.
  This method may throw the following exception:

  - `javax.slee.UnrecognizedServiceException`.
    This exception if thrown if the service specified by the `service` argument is not recognized by the SLEE as a valid Service identifier.

- The `getEventTypes` method.
  This method returns an array of `EventTypeID` objects containing the identifiers of all the event types installed in the SLEE.

- The `getProfileSpecifications` method.
  This method returns an array of `ProfileSpecificationID` objects containing the identifiers of all the Profile Specifications installed in the SLEE.

- The `getServices` method.
  This method returns an array of `ServiceID` objects containing the identifiers of all the Services installed in the SLEE.

- The `getResourceAdaptorTypes` method.
  This method returns an array of `ResourceAdaptorTypeID` objects containing the identifiers of all the resource adaptor types installed in the SLEE.

- The `getResourceAdaptors` method.
  This method returns an array of `ResourceAdaptorID` objects containing the identifiers of all the resource adaptors installed in the SLEE.

- The `getLibraries` method. *(Added in 1.1)*
  This method returns an array of `LibraryID` objects containing the identifiers of all the libraries installed in the SLEE.

- The `getReferringComponents` method.
  This method returns an array of `ComponentID` objects that identify the components that use or

make reference to the component specified by the `id` argument. This method can be used to determine if there are any other installed components depending on the specified component.

This method may throw the following exception:

- `javax.slee.UnrecognizedComponentException`.
  This exception is thrown if the component identifier passed in as an argument is not recognized by the SLEE as a valid identifier.

- The `getDescriptor(DeployableUnit id)` method.
  This method returns a deployable unit descriptor object for the deployable unit identifier specified by the `id` argument.

  This method may throw the following exception:

  - `javax.slee.management.UnrecognizedDeployableUnitException`.
    This exception is thrown if the deployable unit identifier passed in as an argument is not recognized by the SLEE as a valid identifier.

- The `getDescriptors(DeployableUnit[] ids)` method.
  This method performs the function of the above `getDescriptor` method over an array of deployable unit identifiers. The returned array has the same length as the array passed in by the `ids` argument, and if `descriptors = getDescriptors(ids)`, then `descriptors[i] == getDescriptor(ids[i])`. If the `ids` array contains any deployable unit identifier that is unrecognized by the SLEE, the returned array contains a `null` value at the corresponding array index position.

- The `isInstalled(DeployableUnitID id)` method.
  This method determines if the deployable unit identified by the `id` argument is currently installed in the SLEE. It returns `true` if the deployable unit is currently installed in the SLEE, otherwise, it returns `false`.

- The `getDescriptor(ComponentID id)` method.
  This method returns a component descriptor object for the component identifier specified by the `id` argument. The actual type of the descriptor object returned will correspond to the type of the identifier provided. For example, if a `ServiceID` identifier is given as an argument, a `ServiceDescriptor` object will be returned.

  This method may throw the following exception:

  - `javax.slee.UnrecognizedComponentException`.
    This exception is thrown if the component identifier passed in as an argument is not recognized by the SLEE as a valid identifier.

- The `getDescriptors(ComponentID[] ids)` method.
  This method performs the function of the above `getDescriptor` method over an array of component identifiers. The returned array has the same length as the array passed in by the `ids` argument, and if `descriptors = getDescriptors(ids)`, then `descriptors[i] == getDescriptor(ids[i])`. If the `ids` array contains any component identifier that is unrecognized by the SLEE, the returned array contains a `null` value at the corresponding array index position.

- The `isInstalled(ComponentID id)` method.
  This method determines if the component identified by the `id` argument is currently installed in the SLEE. It returns `true` if the component is currently installed in the SLEE, otherwise, it returns `false`.

- The above methods may also throw the following exceptions:

- `javax.slee.management.ManagementException`.
  If the requested operation cannot be performed due to a system-level failure, then the method throws a `ManagementException`. The requested operation may be valid but the SLEE cannot perform the requested operation due to a system-level failure, or the SLEE cannot determine whether the requested operation is valid due to a system-level failure.
- `java.lang.NullPointerException`.
  If any of a method's arguments are `null`, the method throws a `NullPointer-Exception`.

14.7 `ServiceManagementMBean` interface

Changed in 1.1: The JMX Object Name of the ServiceManagementMBean interface is now specified. A Service state change notification has been added.

The `ServiceManagementMBean` interface defines the management API for Services installed in the SLEE. It can be used to:

- Change the operational state of a Service (see Section 2.2.17).
- Obtain the JMX Object Names of `ServiceUsageMBean` objects.

The Object Name of a `ServiceManagementMBean` object is defined by the string "`javax.slee.management:name=ServiceManagement`". *(Added in 1.1)*

The Object Name may also be obtained using the `getServiceManagementMBean` method on a `SleeProviderMBean` object.

Each time the operational state of a service changes, the `ServiceManagementMBean` object generates a service state change notification. The `ServiceStateChangeNotification` class defines the information encapsulated in each service state change notification. The concrete class that implements the `ServiceManagementMBean` interface must extend the `NotificationBroadcaster` interface even though the `ServiceManagementMBean` interface does not extend the `NotificationBroadcaster` interface (see Section 14.3.1).

The `ServiceManagementMBean` interface is as follows:

```
package javax.slee.management;

import javax.management.ObjectName;
import javax.slee.InvalidStateException;
import javax.slee.ServiceID;
import javax.slee.UnrecognizedServiceException;

public interface ServiceManagementMBean {
       // JMX Object Name string of the ServiceManagementMBean
       public static final String OBJECT_NAME
              = "javax.slee.management:name=ServiceManagement";

       // JMX notification type of Service State Change notification
       public static final String SERVICE_STATE_CHANGE_NOTIFICATION_TYPE
              = "javax.slee.management.servicestatechange";

       // methods
       public ServiceState getState(ServiceID id)
              throws  NullPointerException,
                      UnrecognizedServiceException,
                      ManagementException;
       public ServiceID[] getServices(ServiceState state)
              throws  NullPointerException,
                      ManagementException;
       public void activate(ServiceID id)
              throws  NullPointerException,
```

```
                        InvalidStateException,
                        InvalidLinkNameBindingStateException,
                        UnrecognizedServiceException,
                        ManagementException;
        public void activate(ServiceID[] ids)
                throws  NullPointerException,
                        InvalidArgumentException,
                        InvalidStateException,
                        InvalidLinkNameBindingStateException,
                        UnrecognizedServiceException,
                        ManagementException;
        public void deactivate(ServiceID id)
                throws  NullPointerException,
                        InvalidStateException,
                        UnrecognizedServiceException,
                        ManagementException;
        public void deactivate(ServiceID[] ids)
                throws  NullPointerException,
                        InvalidArgumentException,
                        InvalidStateException,
                        UnrecognizedServiceException,
                        ManagementException;
        public void deactivateAndActivate(ServiceID deactivateID,
                                          ServiceID activateID)
                throws  NullPointerException,
                        InvalidArgumentException,
                        InvalidStateException,
                        InvalidLinkNameBindingStateException,
                        UnrecognizedServiceException,
                        ManagementException;
        public void deactivateAndActivate(ServiceID[] deactivateIDs,
                                          ServiceID[] activateIDs)
                throws  NullPointerException,
                        InvalidArgumentException,
                        InvalidStateException,
                        InvalidLinkNameBindingStateException,
                        UnrecognizedServiceException,
                        ManagementException;
        public ObjectName getServiceUsageMBean(ServiceID id)
                throws  NullPointerException,
                        UnrecognizedServiceException,
                        ManagementException;
}
```

- The `OBJECT_NAME` constant. *(Added in 1.1)*
  This constant defines the string form of the `ServiceManagementMBean`'s JMX Object Name.
- The `SERVICE_STATE_CHANGE_NOTIFICATION_TYPE` constant.
  This constant defines the value for the `type` attribute of `ServiceStateChangeNotifications` emitted by a `ServiceManagementMBean` object (refer Section 14.7.2).
- The `getState` method.
  This method returns the current operational state of the Service specified by the `id` argument.
- The `getServices` method.
  This method returns an array of `ServiceID` objects that represent Services that are in the operational state specified by the `state` argument.
- The `activate` methods.
  These methods activate the Service or Services specified by the method argument.
  - It can only be invoked when the specified Service or Services are in the Inactive state.
  - If it returns successfully, the specified Service or Services have transitioned to the Active state.

  These methods may throw the following exceptions.

  - `javax.slee.InvalidArgumentException`.
    This exception is thrown by the array-argument form of the method if the array parameter is zero length or contains `null` or duplicate elements.
  - `javax.slee.InvalidStateException`.
    This exception is thrown if any of the specified Services are not in the Inactive state.
  - `javax.slee.InvalidLinkNameBindingStateException`.
    This exception is thrown if a resource adaptor entity link binding required by an SBB in any of the activating Services is not bound, or is bound to a resource adaptor entity that does not implement the resource adaptor type expected by the SBB.

- The `deactivate` methods.
  These methods deactivate the Service or Services specified by the method argument.
  - It can only be invoked when the specified Service or Services are in the Active state.
  - If it returns successfully, the specified Service or Services have transitioned to the Stopping state. Eventually each deactivated Service will transition to the Inactive state after all SBB entities executing for the Service have completed processing (see Section 2.2.17).

  These methods may throw the following exceptions.
  - `javax.slee.InvalidArgumentException`.
    This exception is thrown by the array-argument form of the method if the array parameter is zero length or contains `null` or duplicate elements.
  - `javax.slee.InvalidStateException`.
    This exception is thrown if the any of the specified Services are not in the Active state.

- The `deactivateAndActivate` methods.
  These method deactivate the Service or Services specified by the first argument and activate the Service or Services specified by the second argument, in a single operation. These methods can be used to facilitate a Service upgrade. For example a currently running Service can be deactivated and drained while a new Service is activated to take over processing of new activities.
  - It can only be invoked when the Service or Services that are to be deactivated are in the Active state, and the Service or Services that are to be activated are in the Inactive state.
  - If it returns successfully, the Service or Services that are to be deactivated have transitioned to the Stopping state, and the Service or Services that are to be activated have transitioned to the Active state.

  These methods may throw the following exceptions.
  - `javax.slee.InvalidArgumentException`.
    This exception is thrown if the same Service is identified to be both deactivated and activated. It is also thrown by the array-argument form of the method if either of the array parameters is zero length or contains `null` or duplicate elements.
  - `javax.slee.InvalidStateException`.
    This exception is thrown if any of the Services identified by the first argument are not in the Active state, or if any of the Services identified by the second argument are not in the Inactive state.
  - `javax.slee.InvalidLinkNameBindingStateException`.
    This exception is thrown if a resource adaptor entity link binding required by an SBB in any of the activating Services is not bound, or is bound to a resource adaptor entity that does not implement the resource adaptor type expected by the SBB.

- The `getServiceUsageMBean` method.
  This method returns the Object Name of a `ServiceUsageMBean` object that can be invoked to obtain the usage information of the Service specified by the `id` argument.
  - It can be invoked when the Service is any operational state.
- The above methods may throw the following exceptions:
  - `javax.slee.UnrecognizedServiceException`.
    If a service identifier argument specifies a Service that is not installed in the SLEE, then the method throws an `UnrecognizedServiceException`.
  - `javax.slee.management.ManagementException`.
    If the requested operation cannot be performed due to a system-level failure, then the method throws a `ManagementException`. The requested operation may be valid but the SLEE cannot perform the requested operation due to a system-level failure, or the SLEE cannot determine whether the requested operation is valid due to a system-level failure.
  - `java.lang.NullPointerException`.
    If any of a method's arguments are `null`, the method throws a `NullPointer-Exception`.

14.7.1 `ServiceState` objects

Changed in 1.1: String representation and fromString method added.

The `ServiceState` class defines an enumerated type for the operational states of a Service. A singleton instance of each enumerated value is guaranteed (via an implementation of the `readResolve` method, see `java.io.Serializable` for more details), so that equality tests using == are always evaluated correctly. For convenience an integer representation of `ServiceState` objects is also available.

The following singleton objects define the valid argument or return values for `ServiceState` objects.

- `INACTIVE`
- `ACTIVE`
- `STOPPING`

The public contents of the `ServiceState` class is as follows:

```
package javax.slee.management;

public final class ServiceState implements java.io.Serializable {
       // singletons
       public static final SleeState INACTIVE …;
       public static final SleeState ACTIVE …;
       public static final SleeState STOPPING …;

       // integer representation
       public static final int SERVICE_INACTIVE …;
       public static final int SERVICE_ACTIVE …;
       public static final int SERVICE_STOPPING …;

       // string representation
       public static final String INACTIVE_STRING = “Inactive”;
       public static final String ACTIVE_STRING = “Active”;
       public static final String STOPPING_STRING = “Stopping”;

       // methods
       public boolean isInactive() { … }
       public boolean isActive() { … }
       public boolean isStopping() { … }
```

```
        public static ServiceState fromString(String state)
                throws NullPointerException, IllegalArgumentException { … }
        public static ServiceState fromInt(int state)
                throws IllegalArgumentException { … }
        public int toInt() { … }
        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
}
```

- Each of the `is<State>` methods determines if this `ServiceState` object represents the `<State>` operational state, and is equivalent to `(this == <STATE>)`. For example, the `isActive` method determines if this `ServiceState` object represents the Active operational state and is equivalent to `(this == ACTIVE)`.
- The `fromInt` and `toInt` methods allow conversion between the `ServiceState` object form and numeric form.
- The `fromInt` method throws a `java.lang.IllegalArgumentException` if the `state` argument is not one of the three integer state representations.
- The `fromString` method allows conversion between `String` and `ServiceState` forms. The conversion is case insensitive. This method throws a `java.lang.NullPointerException` if the argument is null. This method throws a `java.lang.IllegalArgumentException` if the argument is not known to this class.

14.7.2 `ServiceStateChangeNotification` class

Added in 1.1

The public interface of the `ServiceStateChangeNotification` class is as follows:

```
package javax.slee.management;

import javax.management.Notification;
import javax.slee.ServiceID;

public final class ServiceStateChangeNotification extends Notification implements Ven-
dorExtensions {
        // constructor
        public ServiceStateChangeNotification(
                ServiceManagementMBean serviceManagementMBean,
                ServiceID service, ServiceState newState, ServiceState oldState,
                long sequenceNumber) { … }

        // accessors
        public ServiceID getService() { … }
        public ServiceState getNewState() { … }
        public ServiceState getOldState() { … }

        // vendor-specific extension data support
        public static void enableVendorDataSerialization() { … }
        public static void disableVendorDataSerialization() { … }
        public static void enableVendorDataDeserialization() { … }
        public static void disableVendorDataDeserialization() { … }
        public void setVendorData(Object vendorData) { … }
        public Object getVendorData() { … }

        public String toString() { … }
}
```

The `ServiceStateChangeNotification` class inherits all the attributes of the base `Notification` class. Of the inherited attributes, the following attributes have specific meaning defined by the SLEE specification:

- The `type` attribute.
  This attribute specifies the type of the notification being generated. The notification type of all `ServiceStateChangeNotification` objects is defined by `ServiceManagementMBean.SERVICE_STATE_CHANGE_NOTIFICATION_TYPE`.
- The `source` attribute.
  This attribute specifies the `ServiceManagementMBean` that emitted this notification.

The `ServiceStateChangeNotification` class adds the following additional attributes specific to resource adaptor entity state change notifications:

- The `service` attribute.
  This attribute identifies the Service.
- The `newState` attribute.
  This attribute identifies the state that the Service has transitioned to.
- The `oldState` attribute.
  This attribute identifies the previous state the Service was in before the state transition.

The following methods are defined by the `ServiceStateChangeNotification` class:

- The constructor.
  The constructor takes the following arguments:
  - The `source` argument specifies the value of the inherited `source` attribute for the notification.
  - The `service` argument specifies the value of the service attribute for the notification. This identifies the Service that has changed state.
  - The `newState` argument specifies the value of the `newState` attribute for the notification.
  - The `oldState` argument specifies the value of the `oldState` attribute for the notification.
  - The `sequenceNumber` argument specifies the value of the inherited `sequenceNumber` attribute for the notification.
- The `getService` method.
  This method returns the value of the `service` attribute.
- The `getNewState` method.
  This method returns the value of the `newState` attribute.
- The `getOldState` method.
  This method returns the value of the `oldState` attribute.
- The `enableVendorDataSerialization`, `disableVendorDataSerialization`, `enableVendorDataDeserialization`, `disableVendorDataDeserialization`, `setVendorData`, and `getVendorData` methods.
  These methods provide support for vendor-specific data in `ServiceStateChangeNotification` objects. See Section 14.17 for details and an explanation of these methods.
- The `toString` method.
  This method returns a string representation of the `ServiceStateChangeNotification` object.

14.8 `ServiceUsageMBean` interface

Changed in 1.1: The JMX Object Name of this MBean has been specified. A method has been added to provide access to the usage notification manager.

The `ServiceUsageMBean` interface defines the management interface used to interact with usage parameter sets for SBBs in a Service. It defines the methods to create, lookup, and remove usage parameter sets for SBBs in the Service.

This interface can also be used to obtain the JMX Object Name of Usage MBean objects for an SBB (see Section 14.9). Usage MBean objects are used by management clients to query and reset the usage parameters of a single usage parameter set.

Usage parameter sets of an SBB are persistent, i.e. they continue to exist after SLEE restarts. However, the SLEE specification does not specify whether the values of the usage parameters within the usage parameter sets are persistent or durable across SLEE restarts. *(Clarified in 1.1)*

A `ServiceUsageMBean` object serves a particular Service. In other words, it is used to manipulate the usage parameter sets that are maintained by the SLEE for the SBBs in the Service.

The Object Name of a `ServiceUsageMBean` object has the following format: *(Added in 1.1)*

- The domain of the Object Name is "`javax.slee.management.usage`".
- A `type` attribute is defined with the value "`ServiceUsage`".
- The `serviceName`, `serviceVendor`, and `serviceVersion` attributes are defined. The values of these attributes are equal to the name, vendor, and version strings of the service respectively.

As an example, if a particular service has the name "FooService", vendor "FooCompany", and version "1.0", then the Object Name of a `ServiceUsageMBean` for that service would be:

> "`javax.slee.management.usage:type=ServiceUsage,serviceName="FooService", serviceVendor="FooCompany",serviceVersion="1.0`""

The preconstructed Object Name for a `ServiceUsageMBean` for a particular Service may be obtained by invoking the `getServiceUsageMBean` method on a `ServiceManagementMBean` object (see Section 14.5) and passing the Service identifier of the Service as an argument to this method.

The `ServiceUsageMBean` interface is as follows:

```
package javax.slee.management;

import javax.slee.SbbID;
import javax.slee.ServiceID;
import javax.slee.InvalidArgumentException;
import javax.slee.UnrecognizedSbbException;
import javax.slee.usage.UnrecognizedUsageParameterSetNameException;
import javax.management.ObjectName;

public interface ServiceUsageMBean {
        // Base JMX Object Name string of a ServiceUsageMBean
        public static final String BASE_OBJECT_NAME
                = "javax.slee.management.usage:type=ServiceUsage";

        // Object name attribute keys for identifying the Service the
        // ServiceUsageMBean is providing usage information for
        public static final String SERVICE_NAME_KEY = "serviceName";
        public static final String SERVICE_VENDOR_KEY = "serviceVendor";
        public static final String SERVICE_VERSION_KEY = "serviceVersion";

        // methods
        public ServiceID getService()
                throws  ManagementException;
        public void createUsageParameterSet(SbbID id, String paramSetName)
                throws  NullPointerException,
```

```
                        UnrecognizedSbbException,
                        InvalidArgumentException,
                        UsageParameterSetNameAlreadyExistsException,
                        ManagementException;
        public void removeUsageParameterSet(SbbID id, String paramSetName)
                throws  NullPointerException,
                        UnrecognizedSbbException,
                        InvalidArgumentException,
                        UnrecognizedUsageParameterSetNameException,
                        ManagementException;
        public String[] getUsageParameterSets(SbbID id)
                throws  NullPointerException,
                        UnrecognizedSbbException,
                        InvalidArgumentException,
                        ManagementException;
        public ObjectName getSbbUsageMBean(SbbID id)
                throws  NullPointerException,
                        UnrecognizedSbbException,
                        InvalidArgumentException,
                        ManagementException;
        public ObjectName getSbbUsageMBean(SbbID id, String paramSetName)
                throws  NullPointerException,
                        UnrecognizedSbbException,
                        InvalidArgumentException,
                        UnrecognizedUsageParameterSetNameException,
                        ManagementException;
        public ObjectName getSbbUsageNotificationManagerMBean(SbbID id)
                throws  NullPointerException,
                        UnrecognizedSbbException,
                        InvalidArgumentException,
                        ManagementException;
        public void resetAllUsageParameters(SbbID id)
                throws  NullPointerException,
                        UnrecognizedSbbException,
                        InvalidArgumentException,
                        ManagementException;
        public void resetAllUsageParameters()
                throws  ManagementException;
        public void close()
                throws  ManagementException;
}
```

- The `BASE_OBJECT_NAME` constant. *(Added in 1.1)*
  This constant defines the base name that is used to construct a `ServiceUsageMBean`'s JMX Object Name.
- The `SERVICE_NAME_KEY`, `SERVICE_VENDOR_KEY`, and `SERVICE_VERSION_KEY` constants. *(Added in 1.1)*
  These constants define the names of the additional attributes included in the Object Name of a `ServiceUsageMBean`. The value of these attributes in the Object Name depend on the Service the `ServiceUsageMBean` was created for.
- The `getService` method.
  This method returns a `ServiceID` object that identifies the Service that this `ServiceUsageMBean` object represents.
- The `createUsageParameterSet` method.
  Create a new named usage parameter set for the SBB specified by the `id` argument within the service represented by the `ServiceUsageMBean` object with the name specified by the `paramSetName` argument.

  This method may throw the following exceptions:

  - `javax.slee.InvalidArgumentException`.
    This exception is thrown if the `paramSetName` argument is zero-length, or if the SBB

identified by the `id` argument participates in the service represented by the `ServiceUsageMBean` object but does not define a usage parameters interface.

- `javax.slee.management.UsageParameterSetNameAlreadyExistsException`.
  This exception is thrown if the usage parameter set identified by `name` already exists for the SBB identified by the `id` parameter within the service represented by the `ServiceUsageMBean` object.

- The `removeUsageParameterSet` method.
  Remove the named usage parameter set that belongs to the SBB specified by the `id` argument within the Service represented by the `ServiceUsageMBean` object with the name specified by the `paramSetName` argument.

  This method may throw the following exception:

  - `javax.slee.usage.UnrecognizedUsageParameterSetNameException`.
    This exception is thrown if the usage parameter set identified by `name` does not correspond to a usage parameter set that exists for the SBB specified by the `id` parameter within the service represented by the `ServiceUsageMBean` object.

- The `getUsageParameterSets` method.
  This method returns a list containing the names of the named usage parameter sets that belong to the SBB specified by the `id` argument within the Service represented by the `ServiceUsageMBean` object.

- The `getSbbUsageMBean` methods.
  These methods return the Object Name of a `UsageMBean` object for the SBB specified by the `id` argument within the Service represented by the `ServiceUsageMBean` object. The one-argument form of this method returns the Object Name of a `UsageMBean` object that provides management access to the default usage parameter set for the SBB. The two-argument form of this method returns the Object Name of a `UsageMBean` object that provides management access to the specified named usage parameter set for the SBB.

  The two-argument form of this method may throw the following exception:

  - `javax.slee.usage.UnrecognizedUsageParameterSetNameException`.
    This exception is thrown if the usage parameter set identified by `paramSetName` does not correspond to a usage parameter set that exists for the SBB specified by the `id` parameter within the service represented by the `ServiceUsageMBean` object.

- The `getSbbUsageNotificationManagerMBean` method. *(Added in 1.1)*
  This method returns the Object Name of a `UsageNotificationManagerMBean` object that provides management access to the usage notification manager for the SBB specified by the `id` argument within the Service represented by the `ServiceUsageMBean` object.

- The `resetAllUsageParameters` methods.
  These methods reset all usage parameters in the default usage parameter set and all named usage parameters sets. The one-argument form of this method resets the usage parameters of only the SBB specified by the `id` argument argument within the Service represented by the `ServiceUsageMBean` object. The zero-argument form of this method resets the usage parameters of all SBBs within the Service represented by the `ServiceUsageMBean` object. The SLEE sets counter-type usage parameters to zero and removes all samples from sample-type usage parameters.

- The `close` method.
  A management client invokes this method to tell the SLEE that the `ServiceUsageMBean` object is no longer required. The implementation of this method is free to deregister the `Ser-`

viceUsageMBean object from the MBean Server. The management client must assume that the Object Name it had for this ServiceUsageMBean object is no longer valid.

- The above methods may throw the following exceptions:
  - javax.slee.UnrecognizedSbbException.
    The method throws this exception if the id argument does not identify an SBB installed in the SLEE or the SBB is not participating in the Service that is represented by the ServiceUsageMBean object.
  - javax.slee.InvalidArgumentException.
    The method throws this exception if the id argument identifies an SBB installed in the SLEE that is participating in the Service that is represented by the ServiceUsageMBean object but the SBB does not define a usage parameters interface.
  - javax.slee.management.ManagementException.
    If the requested operation cannot be performed due to a system-level failure, then the method throws a ManagementException. The requested operation may be valid but the SLEE cannot perform the requested operation due to a system-level failure, or the SLEE cannot determine whether the requested operation is valid due to a system-level failure.
  - java.lang.NullPointerException.
    If any of a method's arguments are null, the method throws a NullPointerException.

14.9 Usage and Usage Notification Manager MBeans

Changed in 1.1: The SBB Usage MBean interface has been replaced with a more generic Usage MBean interface and the requirements of the interface have changed. The Usage Notification Manager MBean has been added in SLEE 1.1. The type attribute of usage notifications have been changed to take into account that usage notifications may originate from various types of components.

The external management interface for accessing a usage parameter set is defined by the Usage MBean and Usage Notification Manager MBean interfaces. The SLEE generates a Usage MBean interface and a Usage Notification Manager MBean interface for each Usage Parameters interface defined by an SBB, Profile Specification, Resource Adaptor, or internal SLEE subsystem.

Logically, the SLEE maintains an instance of the Usage MBean and Usage Notification Manager MBean interfaces, also known as a Usage MBean object and a Usage Notification Manager MBean object respectively, for each usage parameter set. Using the Usage MBean object, the management client can observe the usage parameters of the usage parameter set and register with the Usage MBean object to receive usage notifications from these usage parameters. Using the Usage Notification Manager MBean object, the management client can enable or disable the generation of usage notifications for selected usage parameters. In many cases a management client may not be interested in receiving usage notifications for some usage parameters. By enabling the generation of usage notifications only for those usage parameters that it is interested in receiving notifications for, a management client reduces the work the SLEE must perform whenever usage parameters are updated.

The UsageNotification class defines the information conveyed in each usage notification. The notification type of a usage notification is determined by the NotificationSource object contained in the Usage MBean object that generated the usage notification (refer Section 14.16 for information on notification sources.) For example, if a Usage MBean object contained an SbbNotification object as the notification source, then the notification type of usage notifications generated by the Usage MBean object is equal to SbbNotification.USAGE_NOTIFICATION_TYPE. Each usage notification is emitted as a UsageNotification object.

Usage notifications must only be emitted by a Usage MBean object after updates to usage parameters have been applied. For example, if a usage notification is emitted by a Usage MBean object to a notification listener, informing it that a new sample has been added to a sample-type usage parameter, management clients must be able to query the Usage MBean object and obtain statistics for that sample-type usage parameter that include the sample value contained in the notification.

A management client can use the SLEE specification defined usage notification filters to filter usage notifications. The SLEE specification defines the following usage notification filters:

- `UsageUpdatedFilter`.
  This allows usage notifications of a particular usage parameter instance to pass.
- `UsageOutOfRangeFilter`.
  This allows usage notifications of a particular usage parameter instance to pass if the value of the usage parameter is outside a given range.
- `UsageThresholdFilter` class.
  This requires the value of a particular usage parameter instance to cross a threshold in either the positive or negative direction before usage notifications relating to the usage parameter instance are passed.

The SLEE generates the Usage MBean interface and its implementation class, the Usage MBean class, when an SBB, Profile Specification, or Resource Adaptor defining a usage parameters interface is deployed.

14.9.1 MBean Object Names

Added in 1.1.

The Object names of Usage MBeans and Usage Notification Manager MBeans are defined by the SLEE specification.

14.9.1.1 Usage MBean Object Name

Added in 1.1.

The Object Name of a `UsageMBean` object has the following format:

- The domain of the Object Name is "`javax.slee.usage`".
- A `type` attribute is defined with the value "`Usage`".
- A `parameterSetName` attribute is defined if the Usage MBean is managing usage information for a named usage parameter set, and its value is the name of the usage parameter set. The value must be quoted (e.g. using the `ObjectName`'s `quote` method) to eliminate the possibility of otherwise illegal characters occuring in the value.
  This attribute is absent in the Object Names of Usage MBeans for the default usage parameter set.
- A `notificationSource` attribute is defined. Additional attributes of the Object Name are defined depending on the value of this attribute:
  - If the value of the `notificationSource` attribute is equal to `SbbNotification.USAGE_NOTIFICATION_TYPE`, then the following additional Object Name attributes are defined:
    - The `serviceName`, `serviceVendor`, and `serviceVersion` attributes.
      The values of these attributes are equal to the name, vendor, and version strings of the SBB's Service component identifier respectively.
    - The `sbbName`, `sbbVendor`, and `sbbVersion` attributes.
      The values of these attributes are equal to the name, vendor, and version strings of the component identifier of the SBB respectively.

- If the value of the `notificationSource` attribute is equal to `ProfileTableNotification.USAGE_NOTIFICATION_TYPE`, then the following additional Object Name attributes are defined:
  - The `profileTableName` attribute.
    The value of this attribute is the name of the profile table.
- If the value of the `notificationSource` attribute is equal to `ResourceAdaptorEntityNotification.USAGE_NOTIFICATION_TYPE`, then the following additional Object Name attribute is defined:
  - The `raEntityName` attribute.
    The value of this attribute is the name of the resource adaptor entity.
- If the value of the `notificationSource` attribute is equal to `SubsystemNotification.USAGE_NOTIFICATION_TYPE`, then the following additional Object Name attribute is defined:
  - The `subsystemName` attribute.
    The value of this attribute is the name of the SLEE internal component or subsystem.

The `SbbNotification`, `ProfileTableNotification`, `ResourceAdaptorEntityNotification`, and `SubsystemNotification` classes each define `getUsageMBeanProperties` methods that will return a string containing the additional Object Name attributes required for the respective notification source, with values that uniquely identify the notification source (refer Section 14.16). This string can be combined with Object Name domain and other Object Name attributes to produce the complete Object Name of the Usage MBean.

As an example, the Object Name of a Usage MBean for the default usage parameter set of an SBB with the name "FooSbb", vendor "FooSbbCompany", and version "1.5", participating in a Service with the name "BarService", vendor "BarCompany", and version "1.0" would be:

"`javax.slee.usage:type=Usage,`
`notificationSource=javax.slee.management.usage.sbb,`
`serviceName="BarService",serviceVendor="BarCompany",`
`serviceVersion="1.0",sbbName="FooSbb",sbbVendor="FooSbbCompany",`
`sbbVersion="1.5"`"

The Object Name of a Usage MBean for the usage parameter set with the name "FooUsage" of a Profile Table with the name "Subscribers" would be:

"`javax.slee.usage:type=Usage,parameterSetName="FooUsage",`
`notificationSource=javax.slee.management.usage.profiletable,`
`profileTableName="Subscribers"`"

The Object Name of a Usage MBean for the usage parameter set with the name "FooUsage" of a resource adaptor entity with the name "JCC" would be:

"`javax.slee.usage:type=Usage,parameterSetName="FooUsage",`
`notificationSource=javax.slee.management.usage.raentity,`
`raEntityName="JCC"`"

The Object Name of a Usage MBean for the default usage parameter set of a SLEE internal subsystem with the name "EventRouter" would be:

"`javax.slee.usage:type=Usage,`
`notificationSource=javax.slee.management.usage.subsystem,`
`subsystemName="EventRouter"`"

A management client may obtain the precomputed Object Name of a Usage MBean for an SBB using the `ServiceUsageMBean` interface. The precomputed Object Name of a Usage MBean for a resource adap-

tor entity may be obtained using the `ResourceManagementMBean` interface. The precomputed Object Name of a Usage MBean for a SLEE internal component or subsystem may be obtained using the `SleeManagementMBean` interface.

14.9.1.2 Usage Notification Manager MBean Object Name

Added in 1.1.

The Object Name of a `UsageNotificationManagerMBean` object is constructed in the same way as the Object Name of a `UsageMBean` object (see Section 14.9.1) with the following differences:

- The `type` attribute has the value "`UsageNotificationManager`".
- The `parameterSetName` attribute is never defined.

14.9.2 Requirements for the Usage MBean interface

The following are the requirements for the Usage MBean interface:

- The Usage MBean interface must include in its definition all the methods defined in the interface `javax.slee.usage.UsageMBean`, either by directly or indirectly inheriting from the `UsageMBean` interface or by including the operations defined in the `UsageMBean` interface in its dynamic interface.
  Note: For backwards compability with SLEE 1.0 management clients, the Usage MBean interface of a Usage MBean for an SBB must also include in its definition all the methods defined in the deprecated `javax.slee.usage.SbbUsageMBean` interface.
- There is corresponding usage parameter accessor method in the Usage MBean interface for each usage parameter name defined in the Usage Parameters interface (see Section 14.9.3).

14.9.3 Requirements for the usage parameter accessor method

The method signatures of the two forms of usage parameter accessor methods are as follows:

```
// counter-type accessor method
public long get<usage parameter name>(boolean reset)
       throws ManagementException;

// sample-type accessor method
public SampleStatistics get<usage parameter name>(boolean reset)
       throws ManagementException;
```

The following are the requirements for each usage parameter accessor method:

- There is a usage parameter accessor method for each usage parameter name declared in the Usage Parameters interface.
- The method name of the accessor method is the lowest-level usage parameter name with the first letter uppercased and prefixed by "get".
- The return type of this accessor method depends on the usage parameter type associated with the usage parameter name.
  - The return type is `long` if the usage parameter name is associated with a counter-type usage parameter.
  - The return type is `javax.slee.usage.SampleStatistics` if the usage parameter name is associated with a sample-type usage parameter.
- The accessor method returns the current value of the usage parameter. The value is a number for a counter-type usage parameter and a `SampleStatistics` object for a sample-type usage parameter.

- If the `reset` argument is true, the SLEE will reset the usage parameter value before the requested operation returns. The returned result is always the pre-reset value of the usage parameter. The SLEE should implement this method as an atomic operation (with respect to concurrent access and updates to the usage parameter being reset.)
- This method may throw a `javax.slee.management.ManagementException` if the requested operation cannot be completed successfully.

14.9.3.1 `SampleStatistics` class

Changed in 1.1: In the SLEE 1.0 specification sample statistics were described by a Java interface. In the SLEE 1.1 specification this interfaces has been converted to a class. This change does not affect existing management client code based solely on the SLEE 1.0 interface. Support for vendor extension data has been added to this class.

The `SampleStatistics` class provides basic statistical information about a sample-type usage parameter.

The `SampleStatistics` class is as follows:

```
package javax.slee.usage;

import java.io.Serializable;
import javax.slee.management.VendorExtensions;

public final class SampleStatistics implements VendorExtensions, Serializable {
        // constructor
        public SampleStatistics() { … }
        public SampleStatistics(long sampleCount, long min, long max, double mean) { … }

        // accessors
        public long getSampleCount() { … }
        public long getMinimum() { … }
        public long getMaximum() { … }
        public double getMean() { … }

        // vendor-specific extension data support
        public static void enableVendorDataSerialization() { … }
        public static void disableVendorDataSerialization() { … }
        public static void enableVendorDataDeserialization() { … }
        public static void disableVendorDataDeserialization() { … }
        public final void setVendorData(Object vendorData) { … }
        public final Object getVendorData() { … }

        public String toString() { … }
}
```

- The zero-argument constructor. *(Added in 1.1)*
  This constructor creates a `SampleStatistics` object representing the case where no samples have been counted yet. The sample count is set to 0, the minimum is set to `Long.MAX_VALUE`, the maximum to `Long.MIN_VALUE`, and the mean to 0.0.
- The four-argument constructor. *(Added in 1.1)*
  This constructor creates a `SampleStatistics` object containing the specified sample data. The constructor takes the following arguments:
  - The `sampleCount` argument specifies the number of samples that have been counted for the usage parameter. When the usage parameter is reset this count returns to zero.
  - The `min` argument specifies the minimum sample value recorded for the usage parameter. If `sampleCount` is zero this value should be `Long.MAX_VALUE`.
  - The `max` argument specifies the maximum sample value recorded for the usage parameter. If `sampleCount` is zero this value should be `Long.MIN_VALUE`.

  - The `mean` argument specifies the mean of the sample values recorded for the usage parameter. If `sampleCount` is zero this value should also be zero.

- The `getSampleCount` method.
  This method returns the number of samples accumulated for the usage parameter.
- The `getMinimum` method.
  This method returns the minimum sample value recorded for the usage parameter.
- The `getMaximum` method.
  This method returns the maximum sample value recorded for the usage parameter.
- The `getMean` method.
  This method returns the mean of the sample values recorded for the usage parameter.
- The `enableVendorDataSerialization`, `disableVendorDataSerialization`, `enableVendorDataDeserialization`, `disableVendorDataDeserialization`, `setVendorData`, and `getVendorData` methods. *(Added in 1.1)*
  These methods provide support for vendor-specific data in `SampleStatistics` objects. See Section 14.17 for details and an explanation of these methods.
- The `toString` method. *(Added in 1.1)*
  This method provides a string representation for the sample statistics.

14.9.4 Requirements for the Usage MBean class

Changed in 1.1: The required version of JMX has changed to 1.2.1.

The SLEE implements a Usage MBean class for each Usage Parameters interface defined by an SBB, a Profile Specification, a Resource Adaptor, or an internal SLEE subsystem. A Usage MBean object allows a management client to access a usage parameter set belonging to a notification source. Each Usage MBean object represents one usage parameter set. The management client invokes the Usage MBean object to access and reset the usage parameters within the usage parameter set. It can also register to receive usage notifications for usage parameters that belong to the usage parameter set.

The following are the requirements for the Usage MBean class:

- The Usage MBean class must be implemented in accordance with the JMX 1.2.1 specification.
- The Usage MBean class can be implemented as any type of MBean supported by the JMX 1.2.1 specification but must present the Usage MBean interface that is constructed using the rules specified in Section 14.9.2 to the MBean Server.
- The Usage MBean class must implement all the methods defined in the `javax.slee.usage.UsageMBean` interface. Section 14.9.5 describes each of these methods.
- For backwards compatibility, if the Usage MBean class is being created for an SBB, then the Usage MBean class must also implement all the methods defined in the `javax.slee.usage.SbbUsageMBean` interface. This interface adds the `getService` and `getSbb` methods, the results of which should be obtained from the corresponding methods on the `SbbNotificationSource` object associated with the Usage MBean. *(Added in 1.1)*
- The Usage MBean class must implement all the usage parameter access methods of the Usage MBean interface.
- As a Usage MBean class can emit notifications, it must implement the JMX `javax.management.NotificationBroadcaster` interface. It must conform to the rules in Section 14.3.1.
- The Usage MBean class may use `javax.slee.management.ManagementException` to report errors to the management client when the requested operation fails due to a system-level problem.

14.9.5 `UsageMBean` interface

All Usage MBean classes implement the `javax.slee.usage.UsageMBean` interface. The `UsageMBean` interface is as follows: *(Changed in 1.1)*

```
package javax.slee.usage;

import javax.slee.InvalidStateException;
import javax.slee.management.NotificationSource;
import javax.slee.management.ManagementException;
import javax.management.ObjectName;

public interface UsageMBean {
        // Base JMX Object Name string of a UsageMBean
        public static final String BASE_OBJECT_NAME = "javax.slee.usage:type=Usage";

        // methods
        public NotificationSource getNotificationSource() throws ManagementException;
        public String getUsageParameterSet() throws ManagementException;
        public ObjectName getUsageNotificationManagerMBean() throws ManagementException;
        public void resetAllUsageParameters() throws ManagementException;
        public void close() throws InvalidStateException, ManagementException;
}
```

- The `BASE_OBJECT_NAME` constant. *(Added in 1.1)*
  This constant defines the base JMX Object Name string of all SLEE Usage MBeans.
- The `getNotificationSource` method. *(Added in 1.1)*
  This method returns a `NotificationSource` object that identifies the SBB, Profile Table, resource adaptor entity, or SLEE subsystem that contains the usage parameter set represented by the Usage MBean object (refer Section 14.16).
- The `getUsageParameterSet` method.
  This method returns the name of the usage parameter set represented by the Usage MBean object. If the Usage MBean object represents the default usage parameter set for the notification source, this method returns `null`. *(Changed in 1.1)*
- The `getUsageNotificationManagerMBean` method. *(Added in 1.1)*
  This method returns the Object Name of a `UsageNotificationManagerMBean` object that can be invoked to enable or disable the generation of usage notifications for each usage parameter managed by the Usage MBean object.
- The `resetAllUsageParameters` method.
  This method resets the values of all the usage parameters within the usage parameter set represented by the Usage MBean object.
- The `close` method.
  A management client invokes this method to tell the SLEE that the Usage MBean object is no longer required. The implementation of this method is free to deregister the Usage MBean object from the MBean Server. The management client must assume that the Object Name it had for this Usage MBean object is no longer valid.

  This method may throw the following exception.

  - `javax.slee.InvalidStateException`.
    This method throws this exception if notification listeners are still attached to the Usage MBean object.

These methods throw a `javax.slee.management.ManagementException` if the requested operation cannot be performed due a system-level failure.

14.9.6 Requirements for the Usage Notification Manager interface

Added in 1.1.

The following are the requirements for the Usage Notification Manager MBean interface:

- The Usage Notification Manager MBean interface must include in its definition all the methods defined in the interface `javax.slee.usage.UsageNotificationManagerMBean`, either by directly or indirectly inheriting from the `UsageNotificationManagerMBean` interface or by including the operations defined in the `UsageNotificationManagerMBean` interface in its dynamic interface.
- For each usage parameter name defined in the Usage Parameters interface there is a managed attributed in the Usage Notification Manager MBean interface which allow the usage notification generation flag for the usage parameter to be accessed or modified (see Section 14.9.7).

14.9.7 Requirements for the usage parameter notification management managed attributes

Added in 1.1.

For each usage parameter name defined in the Usage Parameters interface, the Usage Notification Manager MBean interface defines a read/write managed attribute using the following method signatures:

```
public boolean get<usage-parameter-name>NotificationsEnabled()
       throws ManagementException;
public void set<usage-parameter-name>NotificationsEnabled(boolean enabled)
       throws ManagementException;
```

The following are the requirements for each usage notification management managed attributes:

- There is a managed read/write attribute for each usage parameter name declared in the Usage Parameters interface.
- The method name of the accessor method is the lowest-level usage parameter name with the first letter uppercased, prefixed by “get”, and suffixed with “NotificationsEnabled”.
- The accessor method has no arguments and a return type of `boolean`.
- The accessor method returns the current value of the notifications-enabled flag for the usage parameter.
- The method name of the setter method is the lowest-level usage parameter name with the first letter uppercased, prefixed by “set”, and suffixed with “NotificationsEnabled”.
- The setter method has a single `boolean` argument and no return type.
- The setter method allows the notifications-enabled flag for the usage parameter to be modified.
    - When the notifications-enabled flag for a usage parameter is set to `true`, the generation of usage notifications when the usage parameter is updated is enabled for all usage parameter sets containing the usage parameter, where the notification source of the updated usage parameter is the same notification source as contained in the Usage Notification Manager MBean object.
    - When the notifications-enabled flag for a usage parameter is set to `false`, the generation of usage notifications when the usage parameter is updated is suppressed for all usage parameter sets containing the usage parameter, where the notification source of the updated usage parameter is the same notificaiton source as contained in the Usage Notification Manager MBean object.
- These methods may throw a `javax.slee.management.ManagementException` if the requested operation cannot be completed successfully.

14.9.8 Requirements for the Usage Notification Manager MBean class

Added in 1.1.

The SLEE implements a Usage Notification Manager MBean class for each Usage MBean class that it implements. A Usage Notification Manager MBean object allows a management client to select which usage parameters defined in a Usage Parameters interface generate usage notifications when updated. One Usage Notification Manager MBean object is used to manage the usage notifications generated by a notification source's Usage Parameters interface, regardless of the usage parameter set the usage parameters may reside in. In other words, using a Usage Notification Manager MBean object to enable generation of usage notifications for a particular usage parameter enables the generation of usage notifications for that usage parameter in the default usage parameter set as well as any named usage parameter sets for that notification source.

The following are the requirements for the Usage Notification Manager MBean class:

- The Usage Notification Manager MBean class must be implemented in accordance with the JMX 1.2.1 specification.
- The Usage Notification Manager MBean class can be implemented as any type of MBean supported by the JMX 1.2.1 specification but must present the Usage Notification Manager MBean interface that is constructed using the rules specified in Section 14.9.6 to the MBean Server.
- The Usage Notification Manager MBean class must implement all the methods defined in the `javax.slee.usage.UsageNotificationManagerMBean` interface. Section 14.9.9 describes each of these methods.
- The Usage Notification Manager MBean class must implement all the methods of the Usage MBean interface.

The Usage Notification Manager MBean class may use the `javax.slee.management.ManagementException` to report errors to the management client when the requested operation fails due to a system-level problem.

14.9.9 `UsageNotificationManagerMBean` interface

Added in 1.1.

All Usage Notification Manager MBean classes implement the `javax.slee.usage.UsageNotificationManagerMBean` interface. The `UsageNotificationManagerMBean` interface is as follows:

```
package javax.slee.usage;

import javax.slee.management.NotificationSource;
import javax.slee.management.ManagementException;

public class UsageNotificationManagerMBean {
        // Base JMX Object Name string of a UsageNotificationManagerMBean
        public static final String BASE_OBJECT_NAME
                = "javax.slee.usage:type=UsageNotificationManager";

        // methods
        public NotificationSource getNotificationSource() throws ManagementException;
        public void close() throws ManagementException;
}
```

- The `BASE_OBJECT_NAME` constant.
  This constant defines the base JMX Object Name string of all SLEE Usage Notification Manager MBeans.
- The `getNotificationSource` method.
  This method returns a `NotificationSource` object that identifies the SBB, resource adaptor

entity, or SLEE subsystem that contains the usage parameter set whose notification generation is being managed by this Usage Notification Manager MBean object.

- The `close` method.
  A management client invokes this method to tell the SLEE that the Usage Notification Manager MBean object is no longer required. The implementation of this method is free to deregister the Usage Notification Manager MBean object from the MBean Server. The management client must assume that the Object Name it had for this Usage Notification Manager MBean object is no longer valid.

These methods throw a `javax.slee.management.ManagementException` if the requested operation cannot be performed due a system-level failure.

14.9.10 `UsageNotification` class

Changed in 1.1 to reflect changes to the Usage MBean.

The public interface of the `UsageNotification` class is as follows: *(Changed in 1.1)*

```
package javax.slee.usage;

import javax.slee.SbbID;
import javax.slee.ServiceID;
import javax.slee.management.NotificationSource;
import javax.slee.management.VendorExtensions;
import javax.management.Notification;

public final class UsageNotification extends Notification implements VendorExtensions {
        // constructor
        public UsageNotification(String type, UsageMBean usageMBean,
                NotificationSource notificationSource, String paramSet, String paramName,
                boolean counter, long value, long sequenceNumber, long timestamp){ … }

        // deprecated constructor
        public UsageNotification(SbbUsageMBean sbbUsageMBean, ServiceID serviceID,
                SbbID sbbID, String paramSet, String paramName, boolean counter,
                long value, long sequenceNumber, long timestamp) { … }

        // accessors
        public NotificationSource getNotificationSource() { … }
        public String getUsageParameterSetName() { … }
        public String getUsageParameterName() { … }
        public boolean isCounter() { … }
        public long getValue() { … }

        // vendor-specific extension data support
        public static void enableVendorDataSerialization() { … }
        public static void disableVendorDataSerialization() { … }
        public static void enableVendorDataDeserialization() { … }
        public static void disableVendorDataDeserialization() { … }
        public void setVendorData(Object vendorData) { … }
        public Object getVendorData() { … }

        // deprecated methods
        public ServiceID getService();
        public SbbID getSbb();

        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
}
```

The `UsageNotification` class inherits all the attributes of the base `Notification` class. Of the inherited attributes, the following attributes have specific meaning defined by the SLEE specification: *(Changed in 1.1)*

- The `type` attribute.
  This attribute specifies the type of the usage notification being generated. The type of the notification can be used to infer the type of the `NotificationSource` attribute of the usage notification.
- The `source` attribute.
  This attribute specifies the `UsageMBean` that emitted this notification.

The `UsageNotification` class adds the following additional attributes specific to usage notifications:

- The `notificationSource` attribute. *(Added in 1.1)*
  This attribute provides additional information about the source of the usage parameter set that was updated. The exact type of this attribute can be inferred from the `type` attribute of the usage notification. For example, if the type is equal to `SbbNotification.USAGE_NOTIFICATION_TYPE` then the `notificationSource` attribute can be cast to an `SbbNotification` object to obtain the component identifiers of the service and SBB that defined the usage parameter set.
- The `usageParameterSetName` attribute.
  This attribute specifies the name of the usage parameter set containing the usage parameter that caused this usage notification to the emitted. If the usage parameter resides in the default usage parameter set, this method returns `null`.
- The `usageParameterName` attribute.
  This attribute specified the name of the usage parameter that was updated.
- The `counter` attribute.
  This boolean attributes indicates whether the usage parameter that caused the usage notification to the emitted is a counter-type usage parameter (if `counter` is `true`) or a sample-type usage parameter (if `counter` is `false`).
- The `value` attribute.
  If `counter` is `true`, this attribute provides the accumulated value of the counter-type usage parameter that caused this usage notification to be emitted. If `counter` is `false`, this attribute provides the the sample value of the sample-type usage parameter that caused this usage notification to be emitted. *(Clarified in 1.1)*
- The `service` and `sbb` attributes. *(Deprecated in 1.1)*
  These attributes were defined in the SLEE 1.0 version of this class. They are deprecated in SLEE 1.1 as they were specific to SBBs. The `notificationSource` attribute now identifies the source of a usage notification.

The following methods are defined by the `UsageNotification` class:

- The SLEE 1.1-defined constructor. *(Added in 1.1)*
  This constructor takes the following arguments:
  - The `type` argument specifies the value of the inherited `type` attribute for the notification.
  - The `usageMBean` argument specifies the value of the inherited `source` attribute for the notification.
  - The `notificationSource` argument specifies the value of the `notificationSource` attribute for the notification.
  - The `paramSet` argument specifies the value of the `usageParameterSetName` attribute for the notification.
  - The `paramName` argument specifies the value of the `usageParameterName` attribute for the notification.

  - The `counter` argument specifies the value of the `counter` attribute for the notification.
  - The `value` argument specifies the value of the `value` attribute for the notification.
  - The `sequenceNumber` argument specifies the value of the inherited `sequenceNumber` attribute for the notification.
  - The `timestamp` argument specifies the value of the inherited `timestamp` attribute for the notification.
- The SLEE 1.0-defined constructor. *(Deprecated in 1.1)*
  This constructor takes the following arguments:
  - The `sbbUsageMBean` argument specifies the value of the inherited `source` attribute for the notification.
  - The `serviceID` argument specifies the value of the deprecated `service` attribute for the notification.
  - The `sbbID` argument specifies the value of the deprecated `sbb` attribute for the notification.
  - The `paramSet` argument specifies the value of the `usageParameterSetName` attribute for the notification.
  - The `paramName` argument specifies the value of the `usageParameterName` attribute for the notification.
  - The `counter` argument specifies the value of the `counter` attribute for the notification.
  - The `value` argument specifies the value of the `value` attribute for the notification.
  - The `sequenceNumber` argument specifies the value of the inherited `sequenceNumber` attribute for the notification.
  - The `timestamp` argument specifies the value of the inherited `timestamp` attribute for the notification.
- The `getNotificationSource` method. *(Added in 1.1)*
  This method returns the value of the `notificationSource` attribute. For `UsageNotification` objects created using the SLEE 1.0-defined constructor, this method returns an `SbbNotification` object that encapsulates the deprecated `service` and `sbb` attributes.
- The `getUsageParameterSetName` method.
  This method returns the value of the `usageParameterSetName` attribute.
- The `getUsageParameterName` method.
  This method returns the value of the `usageParameterName` attribute.
- The `isCounter` method.
  This method returns the value of the `counter` attribute.
- The `getValue` method.
  This method returns the value of the `value` attribute
- The `getService` method. *(Deprecated in 1.1)*
  This method returns the value of the deprecated `service` attribute. For `UsageNotification` objects created using the SLEE 1.1-defined constructor whose `notificationSource` attribute is of type `SbbNotification`, this method returns the value of `SbbNotification.getService()`.

- The `getSbb` method. *(Deprecated in 1.1)*
  This method returns the value of the deprecated `sbb` attribute. For `UsageNotification` objects created using the SLEE 1.1-defined constructor whose `notificationSource` attribute is of type `SbbNotification`, this method returns the value of `SbbNotification.getSbb()`.
- The `enableVendorDataSerialization`, `disableVendorDataSerialization`, `enableVendorDataDeserialization`, `disableVendorDataDeserialization`, `setVendorData`, and `getVendorData` methods. *(Added in 1.1)*
  These methods provide support for vendor-specific data in `UsageNotification` objects. See Section 14.17 for details and an explanation of these methods.
- The `equals` method.
  Two `UsageNotification` objects are equivalent, i.e. `usageOne.equals(usageTwo)`, if their `notificationSource`, `usageParameterSetName`, `usageParameterName`, and `counter` attributes are equal, as determined by their corresponding `equals` methods. Two `UsageNotification` objects each created using the deprecated SLEE 1.0 constructor are equivalent if their `service`, `sbb`, `usageParameterSetName`, `usageParameterName`, and `counter` attributes are equal, as determined by their corresponding `equals` methods. Two `UsageNotification` object created using different constructors will never be equal. *(Changed in 1.1)*
- The `hashCode` method.
  This method returns the exclusive or'ed hash codes of the `usageParameterSetName` (if non-null) and `usageParameterName` attributes.
- The `toString` method.
  This method provides a string representation for the usage notification.

14.9.10.1 SLEE 1.0 vs SLEE 1.1 Usage notifications

Added in 1.1.

All Usage MBeans created for SLEE Components installed from using a deployment descriptor containing a DOCTYPE declaration that references a SLEE 1.1-defined DTD must emit a `UsageNotification` object created using the new constructor defined in this release of the specification when required to do so, as determined by the corresponding Usage Notification Manager MBean and the presence of any notification listeners.

For backwards compatibility with existing management clients, a Usage MBean created for an SBB installed from a deployment descriptor containing a `DOCTYPE` declaration that references the SLEE 1.0 SBB jar DTD must instead generate and emit a `UsageNotification` object created with the deprecated SLEE 1.0 constructor.

14.9.11 `UsageUpdatedFilter` class

Changed in 1.1 to support all notification sources defined in 1.1.

The `UsageUpdatedFilter` class allows usage notifications for a selected usage parameter to pass. There are two variants of this filter as described below:

- The first variant uses the `NotificationSource` interface introduced in the SLEE 1.1 specification (see Section 14.16) to determine matching usage notifications. A usage notification may pass a `UsageUpdatedFilter` of this variant if the `notificationSource` and `paramName` arguments to the constructor of this filter class are equal to the `notificationSource` and `usageParameterName` attributes of the usage notification. More precisely, a usage notification, `t`, will pass through this filter if `notificationSource.equals (t.get-`

`NotificationSource())` and `paramName.equals(t.getUsageParameter-Name())`, where `notificationSource` and `paramName` are the constructor parameters for the `UsageNotificationFilter`.

- The second variant is the version originally defined in the SLEE 1.0 specification, and is deprecated in the SLEE 1.1 specification. A usage notification may pass a `UsageUpdatedFilter` of this variant if the `service`, `sbb`, and `paramName` arguments to the constructor of this filter class are equal to the `service`, `sbb`, and `usageParameterName` attributes of the usage notification. More precisely, a usage notification, `t`, will pass through this filter if `service.equals(t.getServiceID())`, `sbb.equals(t.getSbb())`, and `paramName.equals(t.getUsageParameterName())`, where `service`, `sbb`, and `paramName` are the constructor parameters for the `UsageNotificationFilter`.

This filter ignores the usage parameter set the notification comes from.

The public interface of the `UsageUpdatedFilter` class is as follows:

```
package javax.slee.usage;

import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.slee.SbbID;
import javax.slee.ServiceID;
import javax.slee.management.NotificationSource;

public class UsageUpdatedFilter implements NotificationFilter {
        // constructor
        public UsageUpdatedFilter(NotificationSource notificationSource, String paramName)
                { … }

        // deprecated constructor
        public UsageUpdatedFilter(ServiceID service, SbbID sbb, String paramName) { … }

        // methods
        public boolean isNotificationEnabled(Notification notification) { … }
}
```

- The SLEE 1.1-defined constructor. *(Added in 1.1)*
  This constructor must be used to filter `UsageNotification` objects created using the SLEE 1.1-defined `UsageNotification` class constructor (see Section 14.9.10). A `UsageUpdatedFilter` object created with this constructor will completely suppress usage notifications whose `UsageNotification` object was created using the deprecated SLEE 1.0-defined `UsageNotification` class constructor.
  This constructor takes the following arguments:
  - The `notificationSource` argument specifies a `NotificationSource` object a usage notification's `notificationSource` attribute must equal in order for the usage notification to be a candidate to pass through this filter.
  - The `paramName` argument specifies a `String` object a usage notification's `usageParameterName` attribute must equal in order for the usage notification to be a candidate to pass through this filter.
- The SLEE 1.0-defined constructor. *(Deprecated in 1.1)*
  This constructor must be used to filter `UsageNotification` objects created using the deprecated SLEE 1.0-defined `UsageNotification` class constructor (see Section 14.9.10). A `UsageUpdatedFilter` object created with this constructor will completely suppress usage notifications whose `UsageNotification` object was created using the SLEE 1.1-defined `UsageNotification` class constructor.
  This constructor takes the following arguments:

  - The `service` argument specifies a `ServiceID` object a usage notification's service attribute must equal in order for the usage notification to be a candidate to pass through this filter.
  - The `sbb` argument specifies an `SbbID` object a usage notification's service attribute must equal in order for the usage notification to be a candidate to pass through this filter.
  - The `paramName` argument specifies the name of the usage parameter that must have been updated in a usage notification in order for the usage notification to be a candidate to pass through this filter.

- The `isNotificationEnabled` method.
  This method implements the `NotificationFilter` interface method to determine if the `notification` argument satisfies the filter criteria.

14.9.12 `UsageOutOfRangeFilter` class

Changed in 1.1 to support all notification sources in 1.1.

The `UsageOutOfRangeFilter` class allows usage notifications to pass if the value of the specified usage parameter falls outside a given range. This is achieved by comparing the `value` attribute of the usage notification to a low value and a high value. If the value is less than the low value or greater than the high value, the usage notification is allowed to pass through the filter.

There are two variants of this filter as described below:

- The first variant uses the `NotificationSource` interface introduced in the SLEE 1.1 specification (see Section 14.16) to determine matching usage notifications. A usage notification may pass a `UsageOutOfRangeFilter` of this variant if the `notificationSource` and `paramName` arguments to the constructor of this filter class are equal to the `notificationSource` and `usageParameterName` attributes of the usage notification, and the `value` attribute of the usage notification falls outside the range specified by the `lowValue` and `highValue` arguments to the constructor of this filter class.
- The second variant is the version originally defined in the SLEE 1.0 specification, and is deprecated in the SLEE 1.1 specification. A usage notification may pass a `UsageOutOfRangeFilter` of this variant if the `service`, `sbb`, and `paramName` arguments to the constructor of this filter class are equal to the `service`, `sbb`, and `usageParameterName` attributes of the usage notification, and the `value` attribute of the usage notification falls outside the range specified by the `lowValue` and `highValue` arguments to the constructor of this filter class.

This filter ignores the usage parameter set the notification comes from.

The public interface of the `UsageOutOfRangeFilter` class is as follows:

```
package javax.slee.usage;

import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.slee.SbbID;
import javax.slee.ServiceID;
import javax.slee.management.NotificationSource;

public class UsageOutOfRangeFilter implements NotificationFilter {
       // constructor
       public UsageOutOfRangeFilter(NotificationSource notificationSource,
              String paramName, long lowValue, long highValue) { … }

       // deprecated constructor
       public UsageOutOfRangeFilter(ServiceID service, SbbID sbb, String paramName,
              long lowValue, long highValue) { … }
```

```
        // methods
        public boolean isNotificationEnabled(Notification notification) { … }
}
```

- The SLEE 1.1-defined constructor. *(Added in 1.1)*
  This constructor must be used to filter `UsageNotification` objects created using the SLEE 1.1-defined `UsageNotification` class constructor (see Section 14.9.10). A `UsageOutOfRangeFilter` object created with this constructor will completely suppress usage notifications whose `UsageNotification` object was created using the deprecated SLEE 1.0-defined `UsageNotification` class constructor.
  This constructor takes the following arguments:
  - The `notificationSource` argument specifies a `NotificationSource` object a usage notification's `notificationSource` attribute must equal in order for the usage notification to be a candidate to pass through this filter.
  - The `paramName` argument specifies a `String` object a usage notification's `usageParameterName` attribute must equal in order for the usage notification to be a candidate to pass through this filter.
  - The `lowValue` and `highValue` arguments specify the low value and high value respectively of the range used by this filter. If a usage notification's `value` attribute falls outside this specified range, the usage notification may be a candidate to pass through this filter.
- The SLEE 1.0-defined constructor. *(Deprecated in 1.1)*
  This constructor must be used to filter `UsageNotification` objects created using the deprecated SLEE 1.0-defined `UsageNotification` class constructor (see Section 14.9.10). A `UsageOutOfRangeFilter` object created with this constructor will completely suppress usage notifications whose `UsageNotification` object was created using the SLEE 1.1-defined `UsageNotification` class constructor.
  This constructor takes the following arguments:
  - The `service` argument specifies a `ServiceID` object a usage notification's service attribute must equal in order for the usage notification to be a candidate to pass through this filter.
  - The `sbb` argument specifies an `SbbID` object a usage notification's service attribute must equal in order for the usage notification to be a candidate to pass through this filter.
  - The `paramName` argument specifies the name of the usage parameter that must have been updated in a usage notification in order for the usage notification to be a candidate to pass through this filter.
  - The `lowValue` and `highValue` arguments specify the low value and high value respectively of the range used by this filter. If a usage notification's `value` attribute falls outside this specified range, the usage notification may be a candidate to pass through this filter.
- The `isNotificationEnabled` method.
  This method implements the `NotificationFilter` interface method to determine if the `notification` argument satisfies the filter criteria.

14.9.13 `UsageThresholdFilter` class

Changed in 1.1 to include all notification sources in 1.1.

The `UsageThresholdFilter` class allows a usage notification to pass if the value of the specified usage parameter crosses a threshold value in either the positive or negative direction. This is achieved by

comparing the previous value of the usage parameter to the current value of the usage parameter. If the specified threshold value lies between the previous value and the current value then the usage notification is allowed to pass through the filter.

There are two variants of this filter as described below:

- The first variant uses the `NotificationSource` interface introduced in the SLEE 1.1 specification (see Section 14.16) to determine matching usage notifications. A usage notification may pass a `UsageThresholdFilter` of this variant if the `notificationSource` and `paramName` arguments to the constructor of this filter class are equal to the `notificationSource` and `usageParameterName` attributes of the usage notification, and the `value` attribute of the usage notification crosses the threshold value specified by the `threshold` argument to the constructor of this filter class.
- The second variant is the version originally defined in the SLEE 1.0 specification, and is deprecated in the SLEE 1.1 specification. A usage notification may pass a `UsageThresholdFilter` of this variant if the `service`, `sbb`, and `paramName` arguments to the constructor of this filter class are equal to the `service`, `sbb`, and `usageParameterName` attributes of the usage notification, and the `value` attribute of the usage notification crosses the threshold value specified by the threshold argument to the constructor of this filter class.

This filter ignores the usage parameter set the notification comes from.

The public interface of the UsageThresholdFilter class is as follows:

```
package javax.slee.usage;

import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.slee.SbbID;
import javax.slee.ServiceID;
import javax.slee.management.NotificationSource;

public class UsageThresholdFilter implements NotificationFilter {
        // constructor
        public UsageThresholdFilter(NotificationSource notificationSource,
                String paramName, long threshold) { … }

        // deprecated constructor
        public UsageThresholdFilter(ServiceID service, SbbID sbb,
                String paramName, long threshold) { … }

        // methods
        public boolean isNotificationEnabled(Notification notification) { … }
}
```

- The SLEE 1.1-defined constructor. *(Added in 1.1)*
  This constructor must be used to filter `UsageNotification` objects created using the SLEE 1.1-defined `UsageNotification` class constructor (see Section 14.9.10). A `UsageThresholdFilter` object created with this constructor will completely suppress usage notifications whose `UsageNotification` object was created using the deprecated SLEE 1.0-defined `UsageNotification` class constructor.
  This constructor takes the following arguments:
  - The `notificationSource` argument specifies a `NotificationSource` object a usage notification's `notificationSource` attribute must equal in order for the usage notification to be a candidate to pass through this filter.
  - The `paramName` argument specifies a `String` object a usage notification's `usageParameterName` attribute must equal in order for the usage notification to be a candidate to pass through this filter.

  - The `threshold` argument specifies the threshold value used by this filter. If a usage notification's `value` attribute crosses this threshold in either direction, the usage notification may be a candidate to pass through this filter.

- The SLEE 1.0-defined constructor. *(Deprecated in 1.1)*
  This constructor must be used to filter `UsageNotification` objects created using the deprecated SLEE 1.0-defined `UsageNotification` class constructor (see Section 14.9.10). A `UsageThresholdFilter` object created with this constructor will completely suppress usage notifications whose `UsageNotification` object was created using the SLEE 1.1-defined `UsageNotification` class constructor.
  This constructor takes the following arguments:
  - The `service` argument specifies a `ServiceID` object a usage notification's service attribute must equal in order for the usage notification to be a candidate to pass through this filter.
  - The `sbb` argument specifies an `SbbID` object a usage notification's service attribute must equal in order for the usage notification to be a candidate to pass through this filter.
  - The `paramName` argument specifies the name of the usage parameter that must have been updated in a usage notification in order for the usage notification to be a candidate to pass through this filter.
  - The `threshold` argument specifies the threshold value used by this filter. If a usage notification's `value` attribute crosses this threshold in either direction, the usage notification may be a candidate to pass through this filter.
- The `isNotificationEnabled` method.
  This method implements the `NotificationFilter` interface method to determine if the `notification` argument satisfies the filter criteria.

14.10 `ProfileProvisioningMBean` interface

Changed in 1.1: The JMX Object Name of the `ProfileProvisioningMBean` is now specified. New methods have been added to this interface to support dynamic Profile queries and Profile Table usage parameters. A utility method to obtain the names of Profile Tables created from a particular Profile Specification has also been added. Additionally methods which return a `Collection` have been clarified to describe the type of Objects contained in the collection.

The `ProfileProvisioningMBean` interface defines the management API for managing the SLEE's provisioned data. It can be used to:

- Create, delete and rename Profile Tables.
- Add and remove Profiles from Profile Tables.
- Obtain the JMX Object Name of the default Profile for a Profile Table.
- Get the names of all Profile Tables in the SLEE.
- Get the names of all Profiles in a specific Profile Table.
- Obtain the JMX Object Name of a specific Profile within a Profile Table using the unique name of the Profile.
- Obtain the JMX Object Names of a set of Profiles within a Profile Table using an indexed attribute.

The Object Name of a `ProfileProvisioningMBean` object is defined by the string "`javax.slee.management:name=ProfileProvisioning`". *(Added in 1.1)*

The Object Name may also be obtained by invoking the `getProfileProvisioningMBean` method on a `SleeManagementMBean` object.

The `ProfileProvisioningMBean` interface is as follows:

```
package javax.slee.management;

import java.util.Collection
import javax.slee.InvalidArgumentException;
import javax.slee.profile.AttributeNotIndexedException;
import javax.slee.profile.AttributeTypeMismatchException;
import javax.slee.profile.ProfileAlreadyExistsException;
import javax.slee.profile.ProfileID;
import javax.slee.profile.ProfileSpecificationID;
import javax.slee.profile.ProfileTableAlreadyExistsException;
import javax.slee.profile.UnrecognizedProfileNameException;
import javax.slee.profile.UnrecognizedProfileSpecificationException;
import javax.slee.profile.UnrecognizedProfileTableNameException;
import javax.slee.profile.UnrecognizedAttributeException;
import javax.slee.profile.UnrecognizedQueryNameException;
import javax.slee.profile.query.QueryExpression;
import javax.management.ObjectName;

public interface ProfileProvisioningMBean {
        // JMX Object Name string of the ProfileProvisioninMBean
        public static final String OBJECT_NAME
                = "javax.slee.management:name=ProfileProvisioning";

        public void createProfileTable(ProfileSpecificationID id,
                                       String newProfileTableName)
                throws  NullPointerException,
                        InvalidArgumentException,
                        UnrecognizedProfileSpecificationException,
                        ProfileTableAlreadyExistsException,
                        ManagementException;
        public void removeProfileTable(String profileTableName)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        ManagementException;
        public void renameProfileTable(String oldProfileTableName,
                                       String newProfileTableName)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        ProfileTableAlreadyExistsException,
                        InvalidArgumentException,
                        ManagementException;
        public ProfileSpecificationID getProfileSpecification(String profileTableName)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        ManagementException;
        public ObjectName getDefaultProfile(String profileTableName)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        ManagementException;
        public ObjectName createProfile(String profileTableName, String newProfileName)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        InvalidArgumentException,
                        ProfileAlreadyExistsException,
                        ManagementException;
        public void removeProfile(String profileTableName, String profileName)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        UnrecognizedProfileNameException,
                        ManagementException;
        public ObjectName getProfile(String profileTableName, String profileName)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
```

```
                        UnrecognizedProfileNameException,
                        ManagementException;
        public Collection getProfileTables()
                throws  ManagementException;
        public Collection getProfileTables(ProfileSpecificationID id)
                throws  NullPointerException,
                        UnrecognizedProfileSpecificationException,
                        ManagementException;
        public Collection getProfiles(String profileTableName)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        ManagementException;
        public Collection getProfilesByAttribute(String profileTableName,
                        String attributeName, Object attributeValue)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        UnrecognizedAttributeException,
                        InvalidArgumentException,
                        AttributeTypeMismatchException,
                        ManagementException;
        public Collection getProfilesByStaticQuery(String profileTableName,
                                String queryName, Object[] parameters)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        UnrecognizedQueryNameException,
                        InvalidArgumentException,
                        AttributeTypeMismatchException,
                        ManagementException;
        public Collection getProfilesByDynamicQuery(String profileTableName,
                                QueryExpression expr)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        AttributeTypeMismatchException,
                        ManagementException;
        public ObjectName getProfileTableUsageMBean(String profileTableName)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        InvalidArgumentException,
                        ManagementException;

        // deprecated methods
        public Collection getProfilesByIndexedAttribute(String profileTableName,
                        String attributeName, Object attributeValue)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        UnrecognizedAttributeException,
                        AttributeNotIndexedException,
                        AttributeTypeMismatchException,
                        ManagementException;
}
```

- The `OBJECT_NAME` constant. *(Added in 1.1)*
  This constant defines the string form of the `ProfileProvisioningMBean`'s JMX Object Name.
- The `createProfileTable` method.
  This method creates a Profile Table with the name specified by the `newProfileTableName` argument from the Profile Specification specified by the `id` argument.
- The `removeProfileTable` method.
  This method deletes the Profile Table identified by the `profileTableName` argument.
- The `renameProfileTable` method.
  This method renames the Profile Table identified by the `oldProfileTableName` argument to the name specified by the `newProfileTableName` argument.

- The `getProfileSpecification` method.
  This method returns a `ProfileSpecificationID` object that identifies the Profile Specification of the Profile Table specified by the `profileTableName` argument.
- The `getDefaultProfile` method.
  This method gets the JMX Object Name of a Profile MBean object for the default Profile of the Profile Table specified by the `profileTableName` argument. There will always be a default Profile for a Profile Table.
- The `createProfile` method.
  This method gets the JMX Object Name of a modifiable Profile MBean object. When the Profile MBean object commits successfully, the SLEE creates and adds a Profile with the name specified by the `newProfileName` argument to the Profile Table specified by the `profileTableName` argument. Before the Profile MBean object commits, the Profile name specifed by the `newProfileName` argument does not exist and cannot be accessed by SBBs or through the `ProfileProvisioningMBean` interface.
- The `removeProfile` method.
  This method removes the Profile specified by the `profileName` argument from the Profile Table specified by the `profileTableName` argument.
- The `getProfile` method.
  This method gets the JMX Object Name of a Profile MBean object for the Profile identified by the `profileName` argument in the Profile Table specified by the `profileTableName` argument. This method does not provide access to a Profile that is in the process of being created before the Profile MBean object returned by the `createProfile` method corresponding to the Profile commits.
- The `getProfileTables()` method.
  This method returns a `Collection` of `java.lang.String` objects that identify the names of all the Profile Tables that have been created in the SLEE.
- The `getProfileTables(ProfileSpecificationID id)` method. *(Added in 1.1)*
  This method returns a `Collection` of `java.lang.String` objects that identify only the names of the Profile Tables that have been created from the Profile Specification specified by the `id` parameter.
  This method may throw the following exception:
  - `javax.slee.UnrecognizedProfileSpecificationException`.
    This exception if thrown if the Profile Specification specified by the `id` parameter is not recognized by the SLEE as a valid service identifier.

  See Section 14.10.1 for more information on the `Collection` objects returned by `ProfileProvisioningMBean` methods.
- The `getProfiles` method.
  This method returns a `Collection` of `ProfileID` objects that identfy all the Profiles in the Profile Table identified by the `profileTableName` argument. The default Profile of the specified Profile Table is not considered to be in the Profile Table and will not be included in the collection since the default Profile does not have a Profile identifier. See Section 14.10.1 for more information on the `Collection` objects returned by `ProfileProvisioningMBean` methods.
- The `getProfilesByAttribute` method.
  This method returns a `Collection` of `ProfileID` objects that identify the Profiles in the Profile Table identified by the `profileTableName` argument that satisfy the search criteria specified by the `attributeName` and `attributeValue` arguments. The `attributeName` attribute identifies the attribute searched and the `attributeValue` argument identifies the value

to look for in the specified attribute. This method should be used on appropriately indexed attributes. For more information refer to sections 10.21 and 10.22. This method may only be used against Profile Tables created from SLEE 1.1 Profile Specifications. The SLEE throws a `ManagementException` if an attempt is made to invoke this method against a Profile Table created from a SLEE 1.0 Profile Specification.

- If the Java type of the attribute specified by the `attributeName` argument is not an array type, a Profile satisfies the search criteria if the value stored in the specified attribute is equal to the value specified by the `attributeValue` argument.
- The Java type of the attribute is restricted to java primitive types, the Object wrapper for primitive types, `java.lang.String`, or `javax.slee.Address`. If the Java type of the attribute is not a primitive Java type, the Java type of the `attributeValue` argument must be the same as the Java type of the attribute. If the Java type of the attribute is a primitive Java type, the Java type of the `attributeValue` argument must be the Java wrapper class for the primitive type of the attribute.
- The default Profile of the specified Profile Table is not considered to be in the Profile Table and will not be included in the collection since the default Profile does not have a Profile identifier.

See Section 14.10.1 for more information on the `Collection` objects returned by `ProfileProvisioningMBean` methods.

- The `getProfilesByIndexedAttribute` method. *(Deprecated in 1.1)*
  This method returns a `Collection` of `ProfileID` objects that identify the Profiles in the Profile Table identified by the `profileTableName` argument that satisfy the search criteria specified by the `attributeName` and `attributeValue` arguments. The `attributeName` attribute identifies the indexed attribute searched and the `attributeValue` argument identifies the value to look for in the specified attribute. This method has been deprecated in SLEE 1.1. If it is used against a Profile Table created from a SLEE 1.1 Profile Specification the SLEE will reject the invocation by throwing a `ManagementException`.
  - If the Java type of the attribute specified by the `attributeName` argument is not an array type, a Profile satisfies the search criteria if the value stored in the specified attribute is equal to the value specified by the `attributeValue` argument. If the Java type of the indexed attribute is not a primitive Java type, the Java type of the `attributeValue` argument must be the same as the Java type of the indexed attribute. If the Java type of the indexed attribute is a primitive Java type, the Java type of the `attributeValue` argument must be the Java wrapper class for the primitive type of the indexed attribute.
  - If the Java type of the attribute specified by the `attributeName` argument is an array type, a Profile satisfies the search criteria if any element[30] of the array object stored in the specified attribute is equal to the value specified by the `attributeValue` argument. If the Java type of the array element is not a primitive Java type, the Java type of the `attributeValue` argument must be the same as the Java type of the array element. If the Java type of the array element is a primitive Java type, the Java type of the `attributeValue` argument must be the Java wrapper class for the primitive type of the array element.

[30] The term *element* here means a base component of the array that is itself not also an array type. For example, using this terminology, an array of type `Integer[][][]` has elements of type `Integer`. Objects of type `Integer[]` or `Integer[][]` are not considered to be elements of this array for indexing purposes.

- The default Profile of the specified Profile Table is not considered to be in the Profile Table and will not be included in the collection since the default Profile does not have a Profile identifier.

See Section 14.10.1 for more information on the `Collection` objects returned by `ProfileProvisioningMBean` methods.

- The `getProfilesByStaticQuery` method. *(Added in 1.1)*
  This method returns a `Collection` of `ProfileID` objects that identify all the Profiles in the Profile Table identified by the `profileTableName` argument that satisfy the search criteria of the static query identified by the `queryName` argument. The `queryName` argument identifies the search criteria by naming a static query predefined in the deployment descriptor of the profile specification from which the profile table was created. The `parameters` argument is a list of parameter values to apply to the parameters of the static query. The SLEE throws a `ManagementException` if this method is invoked against a Profile Table created from a SLEE 1.0 Profile Specification.
  - The default Profile of the specified Profile Table is not considered to be in the Profile Table and will not be included in the collection since the default Profile does not have a Profile identifier.

  See Section 14.10.1 for more information on the `Collection` objects returned by `ProfileProvisioningMBean` methods.

  This method may throw the following exception:
  - `javax.slee.InvalidArgumentException`.
    This exception is thrown if the number of parameters included in the `parameters` argument is not equal to the number of parameters defined by the query identified by the `queryName` argument.
- The `getProfilesByDynamicQuery` method. *(Added in 1.1)*
  This method returns a `Collection` of `ProfileID` objects that identify the Profiles in the Profile Table identified by the `profileTableName` argument that satisfy the search criteria specified by the `expr` argument. The `expr` argument contains the `attributeName` of the indexed attribute searched, the `attributeValue` to search for in the specified attribute and the operator or combinations of operators to apply to the search criteria. The SLEE throws a `ManagementException` if this method is invoked against a Profile Table created from a SLEE 1.0 Profile Specification.
  - `Equals` match: If the Java type of the `expr` argument is `Equals`, then a Profile will satisfy the search criteria if the Profile attribute identified by the `attributeName` in the `Equals` expression is identical to the `attributeValue` contained in the `Equals` expression.
  - `NotEquals` match: If the Java type of the `expr` argument is `NotEquals`, then a Profile will satisfy the search criteria if the Profile attribute identified by the `attributeName` in the `NotEquals` expression is different to the `attributeValue` contained in the `NotEquals` expression.
  - `GreaterThan` match: If the Java type of the `expr` argument is `GreaterThan`, then a Profile will satisfy the search criteria if the Profile attribute identified by the `attributeName` in the `GreaterThan` expression is greater than the `attributeValue` contained in the `GreaterThan` expression.
  - `GreaterThanOrEquals` match: If the Java type of the `expr` argument is `GreaterThanOrEquals`, then a Profile will satisfy the search criteria if the Profile attribute identified by the `attributeName` in the `GreaterThanOrEquals` expression is

greater than or equal to the `attributeValue` contained in the `GreaterThanOrEquals` expression.

- `LessThan` match: If the Java type of the `expr` argument is `LessThan`, then a Profile will satisfy the search criteria if the Profile attribute identified by the `attributeName` in the `LessThan` expression is less than the `attributeValue` contained in the `LessThan` expression.
- `LessThanOrEquals` match: If the Java type of the `expr` argument is `LessThanOrEquals`, then a Profile will satisfy the search criteria if the Profile attribute identified by the `attributeName` in the `LessThanOrEquals` expression is less than or equal to the `attributeValue` contained in the `LessThanOrEquals` expression.
- `LongestPrefixMatch`: If the Java type of the `expr` argument is `LongestPrefixMatch`, then a Profile will satisfy the search criteria if the Profile attribute identified by the `attributeName` in the `LongestPrefixMatch` expression is the longest prefix match for the prefix string specified by the `attributeValue` contained in the `LongestPrefixMatch` expression. A longest prefix match occurs when no other prefix values belonging to this attribute within the specified Profile Table matches more characters starting from the first character of the `attributeValue` argument. Note: It is possible to have more than one Profile satisfy a longest prefix match query if the attribute within the Profile Table is not unique.
- `HasPrefix` match: If the Java type of the `expr` argument is `HasPrefix`, then a Profile will satisfy the search criteria if the Profile attribute identified by the `attributeName` in the `HasPrefix` expression has a prefix equal to the `attributeValue` contained in the `HasPrefix` expression.
- `RangeMatch`: If the Java type of the `expr` argument is `RangeMatch`, then a Profile will satisfy the search criteria if the Profile attribute identified by the `attributeName` in the `RangeMatch` expression is greater than or equal to the `fromValue` in the `RangeMatch` expression and less than or equal to the `toValue` in the `RangeMatch` expression.
- The `And`, `Or` and `Not` expressions enable composite searches to be carried out on a range of Profile attributes, with the necessary behavior as determined by the respective boolean operators.
- If the Java type of `expr` is not of type `Equals`, `NotEquals`, `GreaterThan`, `GreaterThanOrEquals`, `LessThan`, `LessThanOrEquals`, `LongestPrefixMatch`, `HasPrefix`, `RangeMatch`, `And`, `Or` or `Not` then the behavior of this method is undefined..

- The `getProfileTableUsageMBean` method. *(Added in 1.1)*
  This method returns the Object Name of a `ProfileTableUsageMBean` object that can be invoked to obtain the usage information of the Profile Table specified by the `profileTableName` argument.
  This method may throw the following exceptions:
  - `javax.slee.management.UnrecognizedProfileTableNameException`. This exception is thrown if the Profile Table name passed in as an argument is not recognized by the SLEE as a valid Profile Table name.
  - `javax.slee.InvalidArgumentException`.
    This exception is thrown if the Profile Specification that the specified Profiel Table was created from does not define a usage parameters interface.
- The above methods may throw the following exceptions:

- `javax.slee.InvalidArgumentException`.
  If the `newProfileTableName` argument or the `newProfileName` argument is invalid (see Section 10.2.4), the method throws an `InvalidArgumentException`.

  This exception may also be thrown if there is an attempt to create a new Profile Table with a Profile Table name that is also referenced by an installed Service and the Profile Specification of the create request is different from the Address Profile Specification specified by the root SBB of the Service.

  For example, FooSBB specifies FooProfileSpec as its Address Profile Specification. FooSBB is FooService's root SBB. If FooService specifies FooAddresses as the name of its Address Profile Table, then the Profile Specification of FooAddresses is implicitly bound to FooProfileSpec. If there is an attempt to create a Profile Table named FooAddresses with a different Profile Specification, this exception may be thrown.

  Similarly, this exception may also be thrown if there is an attempt to rename an existing Profile Table to a new Profile Table name that is also referenced by an installed Service and the Profile Specification of the rename request is different from the Address Profile Specification specified by the root SBB of the Service.

  This exception may also be thrown by the `getProfilesByAttribute` method if the type of the identified attribute is not valid for profile queries.
- `javax.slee.profile.UnrecognizedProfileSpecificationException`.
  If the `id` argument does not identify a Profile Specification recognized by the SLEE, the method throws an U`nrecognizedProfileSpecificationException`.
- `javax.slee.profile.UnrecognizedProfileTableNameException`.
  If the `profileTableName` or `oldProfileTableName` argument does not identify a Profile Table recognized by the SLEE, the method throws an U`nrecognizedProfileTableNameException`.
- `javax.slee.profile.UnrecognizedProfileNameException`.
  If the `profileTableName` argument identifies a valid Profile Table but the `profileName` argument does not identify a Profile in the identified Profile Table, then the method throws an `UnrecognizedProfileNameException`.
- `javax.slee.profile.UnrecognizedAttributeException`.
  If the `attributeName` argument does not identify an attribute of the Profile Table specified by the `profileTableName` argument, then the method throws an `UnrecognizedAttributeException`.
- `javax.slee.profile.UnrecognizedQueryNameException`.
  If the `queryName` argument does not identify a query declared in the profile specification deployment descriptor of the Profile Table specified by the `profileTableName` argument, then the method throws an `UnrecognizedQueryNameException`.
- `javax.slee.profile.ProfileTableAlreadyExistsException`.
  If the `newProfileTableName` argument identifies a Profile Table that already exists in the SLEE, the method throws a `ProfileTableAlreadyExistsException`.
- `javax.slee.profile.ProfileAlreadyExistsException`.
  If the `newProfileName` argument identifies a Profile that already exists in the Profile Table identified by the `profileTableName` argument, then the method throws a `ProfileAlreadyExistsException`.

- `javax.slee.profile.AttributeNotIndexedException`.
  If the `attributeName` argument identifies an attribute that is not indexed, then the method throws an `AttributeNotIndexedException`.*)*
- `javax.slee.profile.AttributeTypeMismatchException`.
  If the type of the `attributeValue` argument, or the type of an attribute value in a static or dynamic query does not match the type of the attribute being compared, then the method throws an `AttributeTypeMismatchException`.
- `javax.slee.management.ManagementException`.
  If the requested operation cannot be performed due to a system-level failure, then the method throws a `ManagementException`. The requested operation may be valid but the SLEE cannot perform the requested operation due to a system-level failure, or the SLEE cannot determine whether the requested operation is valid due to a system-level failure.
- `java.lang.NullPointerException`.
  If any of a method's arguments are `null`, the method throws a `NullPointer-Exception`.

See Section 14.10.1 for more information on the `Collection` objects returned by `Profile-ProvisioningMBean` methods.

14.10.1 `Collection` objects returned by `ProfileProvisioningMBean` methods

Clarified in 1.1: Description of the type of Objects contained in each `Collection` *added.*

The `getProfileTables` method returns a `Collection` object that contains `java.lang.String` objects. The `getProfiles`, `getProfilesByIndexedAttribute`, `getProfilesByStatic-Query`, and `getProfilesByDynamicQuery` methods return `Collection` objects that contain `ProfileID` (Profile identifier) objects.

The SLEE specification does not specify whether the `Collection` object returned by one of these methods are mutable. If there are mutations, these mutations have no effect on the SLEE, i.e. the SLEE does not add or remove Profile Tables or Profiles when these `Collection` objects are modified.

The SLEE specification does not specify whether the `Collection` object returned by one of these methods and the `Iterator` objects obtained from the Collection object remain consistent across concurrent modifications to the Profile database that hold the names. For example, if the Profile database is modified after a `Collection` object has been obtained, then the `Collection` object may contain a different set of names than in the Profile database.

Furthermore, if the SLEE implementation does not support concurrent modification, then an `Iterator` object may throw a `ConcurrentModificationException`. An example of when this exception may be thrown is as follows:

1. Invoke `getProfileTables` to get a `Collection` object containing all Profile Table names.
2. Invoke `iterator` on the `Collection` object to obtain an `Iterator` object.
3. Invoke `removeProfileTable` to remove a Profile Table.
4. Invoke `next` to get the next name on the `Iterator` object. If concurrent modification is not supported, `nextName` may throw a `ConcurrentModificationException`.

14.11 `ProfileTableUsageMBean` interface

Added in 1.1.

The `ProfileTableUsageMBean` interface defines the management interface used to interact with usage parameter sets for a Profile Table. It defines the methods to create, lookup, and remove usage parameter sets for the Profile Table.

This interface can also be used to obtain the JMX Object Name of Usage MBean objects for a Profile Table (see Section 14.9). Usage MBean objects are used by management clients to query and reset the usage parameters of a single usage parameter set.

Usage parameter sets of a Profile Table are persistent, i.e. they continue to exist after SLEE restarts. However, the SLEE specification does not specify whether the values of the usage parameters within the usage parameter sets are persistent or durable across SLEE restarts.

A `ProfileTableUsageMBean` object serves a particular Profile Table. In other words, it is used to manipulate the usage parameter sets that are maintained by the SLEE for the Profile Table.

The Object Name of a `ResourceUsageMBean` object has the following format:

- The domain of the Object Name is "`javax.slee.management.usage`".
- A `type` attribute is defined with the value "`ProfileTableUsage`".
- The `profileTableName` attribute is defined. The value of this attribute is equal to the name of the Profile Table.

As an example, if a particular Profile Table has the name "Subscribers" then the Object Name of a `ProfileTableUsageMBean` for that Profile Table would be:

> "`javax.slee.management.usage:type=ProfileTableUsage,raEntityName=Subscribers`"

The preconstructed Object Name for a `ProfileTableUsageMBean` for a particular Profile Tablemay be obtained by invoking the `getProfileTableUsageMBean` method on a `ProfileProvisioningMBean` object (see Section 14.10) and passing the name of the Profile Table as an argument to this method.

The `ProfileTableUsageMBean` interface is as follows:

```
package javax.slee.management;

import javax.management.ObjectName;
import javax.slee.InvalidArgumentException;
import javax.slee.usage.UnrecognizedUsageParameterSetNameException;

public interface ProfileTableUsageMBean {
        // Base JMX Object Name string of a ProfileTableUsageMBean
        public static final String BASE_OBJECT_NAME
                = "javax.slee.management.usage:type=ProfileTableUsage";

        // Object name attribute key for identifying the Profile Table the
        // ProfileTableUsageMBean is providing usage information for
        public static final String PROFILE_TABLE_NAME_KEY = "profileTableName";

        // methods
        public String getProfileTableName();
        public void createUsageParameterSet(String paramSetName)
                throws  NullPointerException,
                        InvalidArgumentException,
                        UsageParameterSetNameAlreadyExistsException,
                        ManagementException;
        public void removeUsageParameterSet(String paramSetName)
                throws  NullPointerException,
                        UnrecognizedUsageParameterSetNameException,
                        ManagementException;
        public String[] getUsageParameterSets()
                throws  ManagementException;
        public ObjectName getUsageMBean()
                throws  ManagementException;
```

```
        public ObjectName getUsageMBean(String paramSetName)
               throws  NullPointerException,
                       UnrecognizedUsageParameterSetNameException,
                       ManagementException;
        public ObjectName getUsageNotificationManagerMBean()
               throws  ManagementException;
        public void resetAllUsageParameters()
               throws  ManagementException;
        public void close()
               throws  ManagementException;
}
```

- The `BASE_OBJECT_NAME` constant.
  This constant defines the base name that is used to construct a `ProfileTableUsageMBean`'s JMX Object Name.
- The `PROFILE_TABLE_NAME_KEY` constant.
  This constant defines the name of the additional attribute included in the Object Name of a `ProfileTableUsageMBean`. The value of this attribute in the Object Name depends on the Profile Table the `ProfileTableUsageMBean` was created for.
- The `getProfileTableName` method.
  This method returns the name of the Profile Table that this `ProfileTableUsageMBean` object represents.
- The `createUsageParameterSet` method.
  Create a new named usage parameter set for the Profile Table represented by the `ProfileTableUsageMBean` object.

  This method may throw the following exceptions:
  - `javax.slee.management.UsageParameterSetNameAlreayExistsException`.
    This exception is thrown if the usage parameter set identified by `name` already exists for the Profile Table represented by the `ProfileTableUsageMBean` object.
- The `removeUsageParameterSet` method.
  Remove the named usage parameter set that belongs to the Profile Table represented by the `ProfileTableUsageMBean` object with the name specified by the `paramSetName` argument.

  This method may throw the following exception:
  - `javax.slee.usage.UnrecognizedUsageParameterSetNameException`.
    This exception is thrown if the usage parameter set identified by `name` does not correspond to a usage parameter set that exists for the Profile Table represented by the `ProfileTableUsageMBean` object.
- The `getUsageParameterSets` method.
  This method returns an array containing the names of the named usage parameter sets that exist for the Profile Table represented by the `ProfileTableUsageMBean` object.
- The `getUsageMBean` methods.
  These methods return the Object Name of a `UsageMBean` object for the Profile Table represented by the `ProfileTableUsageMBean` object. The zero-argument form of this method returns the Object Name of a `UsageMBean` object that provides management access to the default usage parameter set for the Profile Table. The one-argument form of this method returns the Object Name of a `UsageMBean` object that provides management access to the specified named usage parameter set.

  The one-argument form of this method may throw the following exceptions:

  - `javax.slee.usage.UnrecognizedUsageParameterSetNameException`.
    This exception is thrown if the usage parameter set identified by `paramSetName` does not correspond to a usage parameter set that exists for the Profile Table represented by the `ProfileTableUsageMBean` object.

- The `getUsageNotificationManagerMBean` method.
  This method returns the Object Name of a `UsageNotificationManagerMBean` object that provides management access to the usage notification manager for the Profile Table represented by the `ProfileTableUsageMBean` object.
- The `resetAllUsageParameters` methods.
  This method resets all usage parameters in the default usage parameter set and all named usage parameters sets of the Profile Table represented by the `ProfileTableUsageMBean` object. The SLEE sets counter-type usage parameters to zero and removes all samples from sample-type usage parameters.
- The `close` method.
  A management client invokes this method to tell the SLEE that the `ProfileTableUsageMBean` object is no longer required. The implementation of this method is free to deregister the `ProfileTableUsageMBean` object from the MBean Server. The management client must assume that the Object Name it had for this `ProfileTableUsageMBean` object is no longer valid.
- The above methods may throw the following exceptions:
  - `javax.slee.management.ManagementException`.
    If the requested operation cannot be performed due to a system-level failure, then the method throws a `ManagementException`. The requested operation may be valid but the SLEE cannot perform the requested operation due to a system-level failure, or the SLEE cannot determine whether the requested operation is valid due to a system-level failure.
  - `java.lang.NullPointerException`.
    If any of a method's arguments are `null`, the method throws a `NullPointerException`.

14.12 `ResourceManagementMBean` interface

Added in 1.1.

The `ResourceManagementMBean` interface defines the management interface for Resource Adaptors. The `ResourceManagementMBean` interface is an MBean interface that defines operations for managing resource adaptors deployed in the SLEE. The management operations are:

- Create and delete resource adaptor entities.
- Activate and deactivate resource adaptor entities.
- Access resource adaptor and resource adaptor entity configuration properties.
- Update resource adaptor entity configuration properties.
- Bind and unbind link names to resource adaptor entities.
- Access bound link names.
- Access resource adaptor entities based on defined criteria.
- Access the state of resource adaptor entities.
- Obtain the JMX Object Name of `ResourceUsageMBean` objects.

The Object Name of a `ResourceManagementMBean` object is defined by the string "`javax.slee.management:name=ResourceManagement`". The Object Name may also be obtained by invoking the `getResourceManagementMBean` method on a `SleeManagementMBean` object.

Each time the operational state of a resource adaptor entity changes, the `ResourceManagementMBean` object generates a resource adaptor entity state change notification. The `ResourceAdaptorEntityStateChangeNotification` class defines the information encapsulated in each resource adaptor entity state change notification. The concrete class that implements the `ResourceManagementMBean` interface must extend the `NotificationBroadcaster` interface even though the `ResourceManagementMBean` interface does not extend the `NotificationBroadcaster` interface (see Section 14.3.1).

The `ResourceManagementMBean` interface is as follows:

```
package javax.slee.management;

import javax.slee.management.ObjectName;
import javax.slee.InvalidStateException;
import javax.slee.InvalidArgumentException;
import javax.slee.resource.InvalidConfigurationException;
import javax.slee.resource.ResourceAdaptorID;

public interface ResourceManagementMBean {
        // JMX Object Name string of the ResourceManagementMBean
        public static final String OBJECT_NAME =
                "javax.slee.management:name=ResourceManagement";

        // JMX notification type of Resource Adaptor Entity State Change notification
        public static final String RESOURCE_ADAPTOR_ENTITY_STATE_CHANGE_NOTIFICATION_TYPE
                = "javax.slee.management.resourceadaptorentitystatechange";

        public ConfigProperties getConfigurationProperties(ResourceAdaptorID id)
                throws  NullPointerException,
                        UnrecognizedResourceAdaptorException,
                        ManagementException;
        public ConfigProperties getConfigurationProperties(String entityName)
                throws  NullPointerException,
                        UnrecognizedResourceAdaptorEntityException,
                        ManagementException;
        public void createResourceAdaptorEntity(
                                ResourceAdaptorID id, String entityName,
                                ConfigProperties properties)
                throws  NullPointerException,
                        InvalidArgumentException,
                        UnrecognizedResourceAdaptorException,
                        ResourceAdaptorEntityAlreadyExistsException,
                        InvalidConfigurationException,
                        ManagementException;
        public void removeResourceAdaptorEntity(String entityName)
                throws  NullPointerException,
                        UnrecognizedResourceAdaptorEntityException,
                        InvalidStateException,
                        DependencyException,
                        ManagementException;
        public void updateConfigurationProperties(
                                String entityName, ConfigProperties properties)
                throws  NullPointerException,
                        UnrecognizedResourceAdaptorEntityException,
                        InvalidStateException,
                        InvalidConfigurationException,
                        ManagementException;
        public ResourceAdaptorID getResourceAdaptor(String entityName)
                throws  NullPointerException,
                        UnrecognizedResourceAdaptorEntityException,
                        ManagementException;
```

```
        public void activateResourceAdaptorEntity(String entityName)
                throws  NullPointerException,
                        UnrecognizedResourceAdaptorEntityException,
                        InvalidStateException,
                        ManagementException;
        public void deactivateResourceAdaptorEntity(String entityName)
                throws  NullPointerException,
                        UnrecognizedResourceAdaptorEntityException,
                        InvalidStateException,
                        ManagementException;
        public ResourceAdaptorEntityState getState(String entityName)
                throws  NullPointerException,
                        UnrecognizedResourceAdpaterEntityException,
                        ManagementException;
        public String[] getResourceAdaptorEntities()
                throws  ManagementException;
        public String[] getResourceAdaptorEntities(ResourceAdaptorID id)
                throws  NullPointerException,
                        UnrecognizedResourceAdaptorException,
                        ManagementException;
        public String[] getResourceAdaptorEntities(ResourceAdaptorEntityState state)
                throws  NullPointerException,
                        ManagementException;
        public void bindLinkName(String entityName, String linkName)
                throws  NullPointerException,
                        InvalidArgumentException,
                        UnrecognizedResourceAdaptorEntityException,
                        LinkNameAlreadyBoundException,
                        ManagementException;
        public void unbindLinkName(String linkName)
                throws  NullPointerException,
                        UnrecognizedLinkNameException,
                        DependencyException,
                        ManagementException;
        public String[] getLinkNames()
                throws  ManagementException;
        public String[] getLinkNames(String entityName)
                throws  NullPointerException,
                        UnrecognizedResourceAdpaterEntityException,
                        ManagementException;
        public SbbID[] getBoundSbbs(String linkName)
                throws  UnrecognizedLinkNameException,
                        ManagementException;
        public String getResourceAdaptorEntity(String linkName)
                throws  NullPointerException,
                        UnrecognizedLinkNameException,
                        ManagementException;
        public String[] getResourceAdaptorEntities(String[] linkNames)
                throws  NullPointerException,
                        ManagementException;
        public ObjectName getResourceUsageMBean(String entityName)
                throws  NullPointerException,
                        UnrecognizedResourceAdaptorEntityException,
                        InvalidArgumentException,
                        ManagementException;
}
```

- The `OBJECT_NAME` constant.
  This constant defines the string form of the `ResourceManagementMBean`'s JMX Object Name.
- The `RESOURCE_ADAPTOR_ENTITY_STATE_CHANGE_NOTIFICATION_TYPE` constant.
  This constant defines the value for the `type` attribute of `ResourceAdaptorEntityState-ChangeNotification`s emitted by a `ResourceManagementMBean` object (refer Section 14.12.3).

- The `getConfigurationProperties(ResourceAdaptorID id)` method.
  This method returns the `ConfigProperties` object that contains all the configuration properties of the resource adaptor specified by the `id` argument. Properties will be populated with their default values where such values have been defined in the resource adaptor's deployment descriptor.
  This method may throw the following exception:
  - `javax.slee.management.UnrecognizedResourceAdaptorException`.
    This exception is thrown if the resource adaptor passed in as an argument is not recognized by the SLEE as a valid identifier.
- The `getConfigurationProperties(String entityName)` method.
  This method returns a `ConfigProperties` object that contains all the configuration properties and their current values for the resource adaptor entity specified by the `entityName` argument.
  This method may throw the following exception:
  - `javax.slee.management.UnrecognizedResourceAdaptorEntityException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not recognized by the SLEE as a valid resource adaptor entity name.
- The `createResourceAdaptorEntity` method.
  This method creates a new resource adaptor entity from the `id` argument identifying the resource adaptor. The resource adaptor entity is assigned the name specified by the `entityName` argument. A resource adaptor entity is created with a collection of properties specified by the `properties` argument. The `properties` argument must specify values for any properties that do not have a default value defined in the Resource Adaptor deployment descriptor, or a value other than the default is required. When this method returns, the created resource adaptor entity is in the Inactive state.
  This method may throw the following exceptions:
  - `javax.slee.InvalidArgumentException`.
    This exception is thrown if the resource adaptor entity name argument is not a valid. A resource adaptor entity name cannot be `null` or zero-length, and may only contain letter or digit characters as defined by `java.lang.Character.isLetterOrDigit`. Additionally, any other character in the Unicode range 0x0020-0x007e may be included in a resource adaptor entity name.
    This exception is also thrown if the value of any of the configuration properties specified by the `properties` argument is `null`. All configuration properties for a resource adaptor entity must have a non-null value.
  - `javax.slee.management.UnrecognizedResourceAdaptorException`.
    This exception is thrown if the resource adaptor passed in as an argument is not recognized by the SLEE as a valid identifier.
  - `javax.slee.management.ResourceAdaptorEntityAlreadyExistsException`.
    This exception is thrown if the resource adaptor entity name passed in as an argument is already in use by another resource adaptor entity.
  - `javax.slee.management.InvalidConfigurationException`.
    This exception is thrown if any value of any of the configuration properties specified by the `properties` argument is `null`. All configuration properties for a resource adaptor entity must have a non-null value. This exception is also thrown if the `raVerifyConfiguration` method is invoked on a resource adaptor object of the Resource Adaptor with the same configuration properties are specified by the `properties` argument and that method returns by throwing an exception (see Section 15.12.2.1).

- The `removeResourceAdaptorEntity` method.
  This method removes the resource adaptor entity identified by the `entityName` argument. The resource adaptor entity must be in the Inactive state.
  This method may throw the following exceptions:
  - `javax.slee.management.UnrecognizedResourceAdaptorEntityException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not recognized by the SLEE as a valid resource adaptor entity name.
  - `javax.slee.InvalidStateException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not in the Inactive state.
  - `javax.slee.management.DependencyException`.
    This exception is thrown if the resource adaptor entity is still bound to one or more link names.
- The `updateConfigurationProperties` method.
  This method updates the configuration properties of the resource adaptor entity identified by the `entityName` argument. The `properties` argument contains the new configuration parameters, i.e. only the `properties` to be updated need to be specified in this argument.
  The `supports-active-reconfiguration` attribute of the `resource-adaptor-class` element in the Resource Adaptor deployment descriptor determines when resource adaptor entities created from that Resource Adaptor may be reconfigured. If the value of the `supports-active-reconfiguration` attribute is "True", a resource adaptor entity of that Resource Adaptor may be reconfigured when the resource adaptor entity is in any state. If the value of the `supports-active-reconfiguration` attribute is "False", a resource adaptor entity of that Resource Adaptor may be reconfigured only if either the resource adaptor entity is in the Inactive state or the SLEE is in the Stopped state.
  This method may throw the following exceptions:
  - `javax.slee.management.UnrecognizedResourceAdaptorEntityException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not recognized by the SLEE as a valid resource adaptor entity name.
  - `javax.slee.InvalidStateException`.
    This exception is thrown if the value of the `supports-active-reconfiguration` attribute of the `resource-adaptor-class` element in the deployment descriptor of the Resource Adaptor of the resource adaptor entity is `False` and the resource adaptor entity is in the Active or Stopping state and the SLEE is in the Starting, Running, or Stopping state.
  - `javax.slee.management.InvalidConfigurationException`.
    This exception is thrown if any value of any of the configuration properties specified by the `properties` argument is `null`. All configuration properties for a resource adaptor entity must have a non-null value. This exception is also thrown if the `raVerifyConfiguration` method is invoked on a resource adaptor object of the Resource Adaptor with the same configuration properties are specified by the `properties` argument and that method returns by throwing an exception (see Section 15.12.2.1).
- The `getResourceAdaptor` method.
  This method returns the `ResourceAdaptorID` object that identifies the Resource Adaptor from which the resource adaptor entity specified by the `entityName` argument entity was created from.
  This method may throw the following exception:

  - `javax.slee.management.UnrecognizedResourceAdaptorEntityException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not recognized by the SLEE as a valid resource adaptor entity name.

- The `activateResourceAdaptorEntity` method.
  This method changes the operational state of a resource adaptor entity specified by the `entityName` argument to the Active state.
  - Conditions:

    Pre: `ResourceAdaptorMBean.getState(adaptor) == ResourceAdaptorEntityState.INACTIVE`

    Post : `ResourceAdaptorMBean.getState(adaptor) == ResourceAdaptorEntityState.ACTIVE`

  If the SLEE is in the Running state, resource adaptor objects of the resource adaptor entity also transition to the Active state in response to this method invocation.

  This method may throw the following exceptions:
  - `javax.slee.management.UnrecognizedResourceAdaptorEntityException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not recognized by the SLEE as a valid resource adaptor entity name.
  - `javax.slee.InvalidStateException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not in the Inactive state.

- The `deactivateResourceAdaptorEntity` method.
  This method changes the operational state of a resource adaptor entity specified by the `entityName` argument to the Stopping state.
  - Conditions:

    Pre: `ResourceAdaptorMBean.getState(adaptor) == ResourceAdaptorEntityState.ACTIVE`
    Post : `ResourceAdaptorMBean.getState(adaptor) == ResourceAdaptorEntityState.STOPPING`

  If the SLEE is in the Running state, resource adaptor objects of the resource adaptor entity also transition to the Stopping state in response to this method invocation.

  This method may throw the following exceptions:
  - `javax.slee.management.UnrecognizedResourceAdaptorEntityException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not recognized by the SLEE as a valid resource adaptor entity name.
  - `javax.slee.InvalidStateException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not in the Active state.

- The `getState` method.
  This method returns a `ResourceAdaptorEntityState` object that represents the state of the resource adaptor entity specified by the `entityName` argument.
  This method may throw the following exception:
  - `javax.slee.management.UnrecognizedResourceAdaptorEntityException`.

This exception is thrown if the resource adaptor entity passed in as an argument is not recognized by the SLEE as a valid resource adaptor entity name.

- The `getResourceAdaptorEntities()` method.
  This method returns an array of `java.lang.String` objects that name the resource adaptor entities that are currently available in the SLEE.
- The `getResourceAdaptorEntities(ResourceAdaptorID id)` method.
  This method returns an array of `java.lang.String` objects that name the resource adaptor entities that are available in the SLEE which have been created from the Resource Adaptor specified by the `id` argument.
  This method may throw the following exception:
  - `javax.slee.management.UnrecognizedResourceAdaptorException`.
    This exception is thrown if the resource adaptor passed in as an argument is not recognized by the SLEE as a valid identifier.
- The `getResourceAdaptorEntities(ResourceAdaptorEntityState state)` method.
  This method returns an array of `java.lang.String` objects that name the resource adaptor entities that are available in the SLEE which are in the the state specified by the `state` argument.
- The `bindLinkName` method.
  This method binds the resource adaptor entity specified by the `entityName` argument to the link name specified by the `linkName` argument. Link names are used to establish associations between SBB's and resource adaptor entities. An SBB uses the link name in a `resource-adaptor-entity-link` element in its deployment descriptor to bind to a particular resource adaptor entity (refer to Section 6.13.3).
  This method may throw the following exceptions:
  - `javax.slee.InvalidArgumentException`.
    This exception is thrown if the `linkName` argument is not valid. A link name cannot be `null` or zero-length, and may only contain letter or digit characters as defined by `java.lang.Character.isLetterOrDigit`. Additionally, any other character in the Unicode range 0x0020-0x007e may be included in a link name.
  - `javax.slee.management.UnrecognizedResourceAdaptorEntityException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not recognized by the SLEE as a valid resource adaptor entity name.
  - `javax.slee.management.LinkNameAlreadyBoundException`.
    This exception is thrown if the link name passed in as an argument is already bound to a resource adaptor entity. All link names must be unique.
- The `unbindLinkName` method.
  This method unbinds a link name from a resource adaptor entity specified by the `linkName` argument.
  This method may throw the following exceptions:
  - `javax.slee.management.UnrecognizedLinkNameException`.
    This exception is thrown if the link passed in as an argument is not recognized by the SLEE as a valid link name.
  - `javax.slee.management.DependencyException`.
    This exception is thrown if the link name passed in as an argument is still in use by one or more SBBs.

- The `getLinkNames` method.
  This method returns an array containing all the link names currently bound in the SLEE.
- The `getLinkNames(String entityName)` method.
  This method returns an array of link names that are bound to the resource adaptor entity specified by the `entityName` argument.
  This method may throw the following exception:
  - `javax.slee.management.UnrecognizedResourceAdaptorEntityException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not recognized by the SLEE as a valid resource adaptor entity name.
- The `getBoundSbbs` method.
  This method returns an array of SBB Component identifiers that identify the SBBs that are bound to the link name specified by the `linkName` argument by way of a `resource-adaptor-entity-link` element in their deployment descriptor.
  This method may throw the following exception:
  - `javax.slee.management.UnrecognizedLinkNameException`.
    This exception is thrown if the link passed in as an argument is not recognized by the SLEE as a valid link name.
- The `getResourceAdaptorEntity` method.
  This method returns the name of the resource adaptor entity of which the link name specified by the `linkName` argument is bound.
  This method may throw the following exception:
  - `javax.slee.management.UnrecognizedLinkNameException`.
    This exception is thrown if the link name passed in as an argument is not recognized by the SLEE.
- The `getResourceAdaptorEntities(String[] linkNames)` method.
  This method performs the function of the above `getResourceAdaptorEntity` method over an array of link names. The returned array has the same length as the array passed in by the `linkNames` argument, and if `entities = getResourceAdaptorEntities(linkNames)`, then `entities[i] == getResourceAdaptorEntiy(linkNames[i])`. If the array contains any link name that is unrecognized by the SLEE, the returned array contains a `null` value at the corresponding array index position.
- The `getResourceUsageMBean` method.
  This method returns the Object Name of a `ResourceUsageMBean` object that can be invoked to obtain the usage information of the resource adaptor entity specified by the `entityName` argument.
  This method may throw the following exceptions:
  - `javax.slee.management.UnrecognizedResourceAdaptorEntityException`.
    This exception is thrown if the resource adaptor entity passed in as an argument is not recognized by the SLEE as a valid resource adaptor entity name.
  - `javax.slee.InvalidArgumentException`.
    This exception is thrown if the resource adaptor that the specified resource adaptor entity was created from does not define a usage parameters interface.
- The above methods may also throw the following exceptions:

- `java.lang.NullPointerException`.
  If any of a method's arguments are `null`, the method throws a `NullPointer-Exception`.
- `javax.slee.management.ManagementException`.
  If the requested operation cannot be performed due to a system-level failure, then the method throws a `ManagementException`. The requested operation may be valid but the SLEE cannot perform the requested operation due to a system-level failure, or the SLEE cannot determine whether the requested operation is valid due to a system-level failure.

14.12.1 `ConfigProperties` objects

Added in 1.1.

The configuration properties of Resource Adaptors and resource adaptor entities are reflected in `javax.slee.resource.ConfigProperties` objects. A `ConfigProperties` object is a set of type-safe `javax.slee.resource.ConfigProperties.Property` objects. One `Property` object is used for each configuration property. Configuration properties are declared by a Resource Adaptor Developer, or a SLEE vendor. Resource Adaptors declare configuration properties in their deployment descriptor. When a configuration property is declared in the deployment descriptor, an optional value default can be specified. `Property` objects contained in a `ConfigProperties` object obtained for a Resource Adaptor will contain the default values for configuration properties where they have been declared. Configuration properties that have no specified default value will contain a null value in the corresponding `Property` object. When a resource adaptor entity is created, all `Property` objects must have non-null values.

Configuration property names that start with "`javax.slee:`" are reserved for the SLEE specification. The following configuration properties are defined by this specification:

- "`javax.slee:name`". This is of type `java.lang.String`. It is the name of the SLEE implementation. It may be a product name.
- "`javax.slee:vendor`". This is of type `java.lang.String`. It is the vendor of the SLEE implementation.
- "`javax.slee:version`". This is of type `java.lang.String`. It is the version of the SLEE implementation.

Values for each of the above properties are added by the SLEE to a `ConfigProperties` object before the `ConfigProperties` object is passed to a resource adaptor object.

A SLEE vendor may provide configuration properties to a Resource Adaptor. In such a case the property name should be prefixed with "`slee-vendor:`". Additionally such property names should include utilize Java packaging conventions[31]. SLEE vendor defined configuration properties are not subject to the standard Java type requirements for configuration properties if the configuration property is defined only within the scope of the JVM. For example, the SLEE may provide resource adaptors with a reference to a proprietary extension facility via a configuration property. Such an object reference is only valid within the JVM and the configuration property would not be exposed to JMX management clients. If the SLEE may expose a vendor-defined configuration property to external management clients, then the standard type requirements still apply.

A Resource Adaptor should be programmed to ignore SLEE vendor defined properties that it does not recognize.

Example properties are:

[31] E.g. properties from a SLEE vendor organization "mycompany" should be prefixed with "`slee-vendor:com.mycompany.`"

- ```
  ConfigProperties.Property myProperty1 =
        new ConfigProperties.Property(
              "host", "java.lang.String", "localhost"
        );
  ```
- ```
  ConfigProperties.Property myProperty2 =
        new ConfigProperties.Property(
              "port", "java.lang.Integer", new Integer(5060)
        );
  ```

The public interface of the `ConfigProperties` class is defined as follows:

```
package javax.slee.management;

import java.io.Serializable;

public final class ConfigProperties implements Serializable, Cloneable {
       // constructors
       public ConfigProperties() { … }
       public ConfigProperties(ConfigProperties.Property[] properties)
              throws NullPointerException, IllegalArgumentException { … }

       // methods
       public ConfigProperties.Property[] getProperties() { … }
       public ConfigProperties.Property getProperty(String name) { … }
       public void addProperty(ConfigProperties.Property property)
              throws  NullPointerException, IllegalArgumentException { … }

       public String toString() { … }
       public Object clone() { … }
}
```

- The zero-argument constructor.
  This constructor creates a `ConfigProperties` object that initially contains no configuration properties.
- The one-argument constructor.
  This constructor creates a `ConfigProperties` object that is initially populated with the configuration properties specified by the `properties` argument.
  This constructor may throw the following exceptions:
  - `java.lang.NullPointerException`.
    If the `properties` argument is `null`, or contains `null` elements, the constructor throws a `NullPointerException`.
  - `java.lang.IllegalArgumentException`.
    If the `poperties` argument contains configuration properties with the same name, the constructor throws an `IllegalArgumentException`.
- The `getProperties` method.
  This method returns an array representing the collection of properties contained within this `ConfigProperties` object. The properties are of the SLEE defined type `ConfigProperties.Property`.
- The `getProperty` method.
  This method returns the property identified by the `name` argument. If the property does not exist in this `ConfigProperties` object this method returns `null`.
- The `addProperty` method.
  This method adds a new property to the `ConfigProperties` object.
  This method may throw the following exceptions:

  - `java.lang.NullPointerException`.
    If the `property` argument is `null`, the method throws a `NullPointerException`.
  - `java.lang.IllegalArgumentException`.
    If a property with the same name already exists in this `ConfigProperties` object, the method throws an `IllegalArgumentException`.

- The `toString` method.
  This method returns a string representation for the `ConfigProperties` object.
- The `clone` method.
  This method implements the `Cloneable` interface. It returns a deep copy of this `ConfigProperties` object. Changes can be made to the `ConfigProperties` object returned from this method, or to the `Property` objects contained in it, without affecting the original `ConfigProperties` object.

14.12.1.1 `ConfigProperties.Property` objects

Added in 1.1.

A `ConfigProperties.Property` object represents a configurable characteristic of a Resource Adaptor, such an IP address or a port number.

The public interface of the `ConfigProperties.Property` class is defined as follows:

```
package javax.slee.management;

import java.io.Serializable;

public final class ConfigProperties.Property implements Serializable {
        // constructor
        public ConfigProperties.Property(String name, String type, Object value)
                throws NullPointerException, IllegalArgumentException { … }

        // methods
        public String getName() { … }
        public String getType() { … }
        public Object getValue() { … }
        public void setValue(Object value) throws IllegalArgumentException { … }

        public static Object toObject(String type, String value)
                throws IllegalArgumentException { … }

        public boolean equals(Object obj) {….}
        public int hashcode() {….};
        public String toString() {….}
}
```

- The constructor.
  The constructor takes the following arguments:
  - The `name` argument specifies the name of the configuration property.
  - The `type` argument specifies the type of the configuration property. The allowable types for configuration properties are: `java.lang.Integer`, `java.lang.Long`, `java.lang.Double`, `java.lang.Float`, `java.lang.Short`, `java.lang.Byte`, `java.lang.Character`, `java.lang.Boolean`, and `java.lang.String`. These type requirements do not apply to SLEE vendor defined configuration properties defined only within the scope of the SLEE JVM (see Section 14.12.1).
  - The `value` attribute specifies the optional value for the configuration property. The argument may be `null` if no value is specified.

- The `getName` method.
  This method returns the `name` argument of the constructor.
- The `getType` method.
  This method returns the `type` argument of the constructor.
- The `getValue` method.
  This method returns the current value of the configuration property. The returned value can be cast to the type specified by the `getType` method. If no value has been set for the property, this method returns `null`.
- The `setValue` method.
  This method sets the value of this configuration property to value identified by the `value` argument.
  This method may throw the following exception:
  - `java.lang.IllegalArgumentException`.
    If the type of the `value` argument is not the same as the type of this configuration property, as determined by the `getType` method.
- The `toObject` method.
  This utility method converts the string representation of a value to a value object of a specified type. For example `toObject("java.lang.Integer","5")` returns a `java.lang.Integer` object containing the value `5.`

  This method may throw the following exceptions:
  - `java.lang.NullPointerException`.
    If any of a method's arguments are `null`, the method throws a `NullPointerException`.
  - `java.lang.IllegalArgumentException`.
    If the `type` argument is not supported, or the value specified is not compatible with the specified type, this method throws an `IllegalArgumentException`.
- The `equals` method.
  This method indicates whether the specified object (specified by the `obj` argument) is equal to this configuration property. The specified object is equal to this `Property` object if the specified object is a `Property` object and the name, value and type in the specified `Property` object are the same as this Property object.
- The `hashcode` method.
  This method provides a reasonable hash code value for the configuration property based on the property name.
- The `toString` method.
  This method returns a string representation of the configuration property.

14.12.2 `ResourceAdaptorEntityState` objects

Added in 1.1.

The `ResourceEntityAdaptorState` class defines an enumerated type for the operational states of a resource adaptor entity. A singleton instance of each enumerated value is guaranteed (via an implementation of the `readResolve` method, see `java.io.Serializable` for more details), so that equality tests using == are always evaluated correctly. For convenience, an integer representation of `ResourceAdaptorEntityState` objects is also available.

The following singleton objects define the valid argument or return values for `ResourceAdaptorEntityState` objects.

- `INACTIVE`
- `ACTIVE`
- `STOPPING`

The public interface of the `ResourceAdaptorEntityState` class is as follows:

```
package javax.slee.management;

public final class ResourceAdaptorEntityState implements java.io.Serializable {
        // singletons
        public static final ResourceAdaptorEntityState INACTIVE …;
        public static final ResourceAdaptorEntityState ACTIVE …;
        public static final ResourceAdaptorEntityState STOPPING …;

        // integer representation
        public static final int ENTITY_INACTIVE …;
        public static final int ENTITY_ACTIVE …;
        public static final int ENTITY_STOPPING …;

        // string representation
        public static final String INACTIVE = “Inactive”;
        public static final String ACTIVE = “Active”;
        public static final String STOPPING = “Stopping”;

        // methods
        public boolean isInactive() { … }
        public boolean isActive(){ … }
        public boolean isStopping() { … }

        public static ResourceAdaptorEntityState fromString(String state)
                throws NullPointerException, IllegalArgumentException { … }
        public static ResourceAdaptorEntityState fromInt(int state)
                throws IllegalArgumentException { … }
        public int toInt() { … }
        public boolean equals(Object obj) { … }
        public int hashCode(){ … }
        public String toString(){ … }
}
```

- Each of the `is<State>` methods determines if this `ResourceAdaptorEntityState` object represents the `<State>` operational state, and is equivalent to `(this == <STATE>)`. For example, the `isActive` method determines if this `ResourceAdaptorEntityState` object represents the Active operational state and is equivalent to `(this == ResourceAdaptorEntityState.ACTIVE)`.
- The `fromInt` and `toInt` methods allow conversion between the `ResourceAdaptorEntityState` object form and numeric form.
- The `fromInt` method throws a `java.lang.IllegalArgumentException` if the `state` argument is not one of the three integer state representations.
- The `fromString` method allows conversion between `String` and `ResourceAdaptorEntity` forms. The conversion is case insensitive. This method throws a `java.lang.NullPointerException` if the argument is `null`. This method throws a `java.lang.IllegalArgumentException` if the argument is not known to this class.

14.12.3 `ResourceAdaptorEntityStateChangeNotification` class

Added in 1.1

The public interface of the `ResourceAdaptorEntityStateChangeNotification` class is as follows:

```
package javax.slee.management;

import javax.management.Notification;

public final class ResourceAdaptorEntityStateChangeNotification extends Notification im-
plements VendorExtensions {
        // constructor
        public ResourceAdaptorEntityStateChangeNotification(
                ResourceManagementMBean resourceManagementMBean,
                String entityName, ResourceAdaptorEntityState newState,
                ResourceAdaptorEntityState oldState, long sequenceNumber) { … }

        // accessors
        public String getEntityName() { … }
        public ResourceAdaptorEntityState getNewState() { … }
        public ResourceAdaptorEntityState getOldState() { … }

        // vendor-specific extension data support
        public static void enableVendorDataSerialization() { … }
        public static void disableVendorDataSerialization() { … }
        public static void enableVendorDataDeserialization() { … }
        public static void disableVendorDataDeserialization() { … }
        public void setVendorData(Object vendorData) { … }
        public Object getVendorData() { … }

        public String toString() { … }
}
```

The `ResourceAdaptorEntityStateChangeNotification` class inherits all the attributes of the base `Notification` class. Of the inherited attributes, the following attributes have specific meaning defined by the SLEE specification:

- The `type` attribute.
  This attribute specifies the type of the notification being generated. The notification type of all `ResourceAdaptorEntityStateChangeNotification` objects is defined by `ResourceManagement-MBean.RESOURCE_ADAPTOR_ENTITY_STATE_CHANGE_NOTIFICATION_TYPE`.
- The `source` attribute.
  This attribute specifies the `ResourceManagementMBean` that emitted this notification.

The `ResourceAdaptorEntityStateChangeNotification` class adds the following additional attributes specific to resource adaptor entity state change notifications:

- The `entityName` attribute.
  This attribute identifies the name of the resource adaptor entity.
- The `newState` attribute.
  This attribute identifies the state that the resource adaptor entity has transitioned to.
- The `oldState` attribute.
  This attribute identifies the previous state the resource adaptor entity was in before the state transition.

The following methods are defined by the `ResourceAdaptorEntityStateChangeNotification` class:

- The constructor.
  The constructor takes the following arguments:
  - The `source` argument specifies the value of the inherited `source` attribute for the notification.
  - The `entityName` argument specifies the value of the `entityName` attribute for the notification. This is the name of the resource adaptor entity that has changed state.

  - The `newState` argument specifies the value of the `newState` attribute for the notification.
  - The `oldState` argument specifies the value of the `oldState` attribute for the notification.
  - The `sequenceNumber` argument specifies the value of the inherited `sequenceNumber` attribute for the notification.

- The `getEntityName` method.
  This method returns the value of the `entityName` attribute.
- The `getNewState` method.
  This method returns the value of the `newState` attribute.
- The `getOldState` method.
  This method returns the value of the `oldState` attribute.
- The `enableVendorDataSerialization`, `disableVendorDataSerialization`, `enableVendorDataDeserialization`, `disableVendorDataDeserialization`, `setVendorData`, and `getVendorData` methods.
  These methods provide support for vendor-specific data in `ResourceAdaptorEntityStateChangeNotification` objects. See Section 14.17 for details and an explanation of these methods.
- The `toString` method.
  This method returns a string representation of the `ResourceAdaptorEntityStateChangeNotification` object.

14.13 `ResourceUsageMBean` interface

Added in 1.1.

The `ResourceUsageMBean` interface defines the management interface used to interact with usage parameter sets for a resource adaptor entity. It defines the methods to create, lookup, and remove usage parameter sets for the resource adaptor entity.

This interface can also be used to obtain the JMX Object Name of Usage MBean objects for a resource adaptor entity (see Section 14.9). Usage MBean objects are used by management clients to query and reset the usage parameters of a single usage parameter set.

Usage parameter sets of a resource adaptor entity are persistent, i.e. they continue to exist after SLEE restarts. However, the SLEE specification does not specify whether the values of the usage parameters within the usage parameter sets are persistent or durable across SLEE restarts.

A `ResourceUsageMBean` object serves a particular resource adaptor entity. In other words, it is used to manipulate the usage parameter sets that are maintained by the SLEE for the resource adaptor entity.

The Object Name of a `ResourceUsageMBean` object has the following format:

- The domain of the Object Name is "`javax.slee.management.usage`".
- A `type` attribute is defined with the value "`ResourceUsage`".
- The `raEntityName` attribute is defined. The value of this attribute is equal to the name of the resource adaptor entity.

As an example, if a particular resource adaptor entity has the name "FooResource" then the Object Name of a `ResourceUsageMBean` for that resource adaptor entity would be:

> "`javax.slee.management.usage:type=ResourceUsage,raEntityName=FooResource`"

The preconstructed Object Name for a `ResourceUsageMBean` for a particular resource adaptor entity may be obtained by invoking the `getResourceUsageMBean` method on a `ResourceManagementMBean` object (see Section 14.12) and passing the name of the resource adaptor entity as an argument to this method.

The `ResourceUsageMBean` interface is as follows:

```
package javax.slee.management;

import javax.management.ObjectName;
import javax.slee.InvalidArgumentException;
import javax.slee.usage.UnrecognizedUsageParameterSetNameException;

public interface ResourceUsageMBean {
        // Base JMX Object Name string of a ResourceUsageMBean
        public static final String BASE_OBJECT_NAME
                = "javax.slee.management.usage:type=ResourceUsage";

        // Object name attribute key for identifying the resource adaptor entity the
        // ResourceUsageMBean is providing usage information for
        public static final String RESOURCE_ADAPTOR_ENTITY_NAME_KEY = "raEntityName";

        // methods
        public String getEntityName();
        public void createUsageParameterSet(String paramSetName)
                throws  NullPointerException,
                        InvalidArgumentException,
                        UsageParameterSetNameAlreadyExistsException,
                        ManagementException;
        public void removeUsageParameterSet(String paramSetName)
                throws  NullPointerException,
                        UnrecognizedUsageParameterSetNameException,
                        ManagementException;
        public String[] getUsageParameterSets()
                throws  ManagementException;
        public ObjectName getUsageMBean()
                throws  ManagementException;
        public ObjectName getUsageMBean(String paramSetName)
                throws  NullPointerException,
                        UnrecognizedUsageParameterSetNameException,
                        ManagementException;
        public ObjectName getUsageNotificationManagerMBean()
                throws  ManagementException;
        public void resetAllUsageParameters()
                throws  ManagementException;
        public void close()
                throws  ManagementException;
}
```

- The `BASE_OBJECT_NAME` constant.
  This constant defines the base name that is used to construct a `ResourceUsageMBean`'s JMX Object Name.
- The `RESOURCE_ADAPTOR_ENTITY_NAME_KEY` constant.
  This constant defines the name of the additional attribute included in the Object Name of a `ResourceUsageMBean`. The value of this attribute in the Object Name depends on the resource adaptor entity the `ResourceUsageMBean` was created for.
- The `getEntityName` method.
  This method returns the name of the resource adaptor entity that this `ResourceUsageMBean` object represents.
- The `createUsageParameterSet` method.
  Create a new named usage parameter set for the resource adaptor entity represented by the `ResourceUsageMBean` object.

This method may throw the following exceptions:

- `javax.slee.management.UsageParameterSetNameAlreayExistsException`.
  This exception is thrown if the usage parameter set identified by `name` already exists for the resource adaptor entity represented by the `ResourceUsageMBean` object.

- The `removeUsageParameterSet` method.
  Remove the named usage parameter set that belongs to the resource adaptor entity represented by the `ResourceUsageMBean` object with the name specified by the `paramSetName` argument.

  This method may throw the following exception:

  - `javax.slee.usage.UnrecognizedUsageParameterSetNameException`.
    This exception is thrown if the usage parameter set identified by `name` does not correspond to a usage parameter set that exists for the resource adaptor entity represented by the `ResourceUsageMBean` object.

- The `getUsageParameterSets` method.
  This method returns an array containing the names of the named usage parameter sets that exist for the resource adaptor entity represented by the `ResourceUsageMBean` object.

- The `getUsageMBean` methods.
  These methods return the Object Name of a `UsageMBean` object for the resource adaptor entity represented by the `ResourceUsageMBean` object. The zero-argument form of this method returns the Object Name of a `UsageMBean` object that provides management access to the default usage parameter set for the resource adaptor entity. The one-argument form of this method returns the Object Name of a `UsageMBean` object that provides management access to the specified named usage parameter set.

  The one-argument form of this method may throw the following exceptions:

  - `javax.slee.usage.UnrecognizedUsageParameterSetNameException`.
    This exception is thrown if the usage parameter set identified by `paramSetName` does not correspond to a usage parameter set that exists for the resource adaptor entity represented by the `ResourceUsageMBean` object.

- The `getUsageNotificationManagerMBean` method.
  This method returns the Object Name of a `UsageNotificationManagerMBean` object that provides management access to the usage notification manager for the resource adaptor entity represented by the `ResourceUsageMBean` object.

- The `resetAllUsageParameters` methods.
  This method resets all usage parameters in the default usage parameter set and all named usage parameters sets of the resource adaptor entity represented by the `ResourceUsageMBean` object. The SLEE sets counter-type usage parameters to zero and removes all samples from sample-type usage parameters.

- The `close` method.
  A management client invokes this method to tell the SLEE that the `ResourceUsageMBean` object is no longer required. The implementation of this method is free to deregister the `ResourceUsageMBean` object from the MBean Server. The management client must assume that the Object Name it had for this `ResourceUsageMBean` object is no longer valid.

- The above methods may throw the following exceptions:

  - `javax.slee.management.ManagementException`.
    If the requested operation cannot be performed due to a system-level failure, then the method throws a `ManagementException`. The requested operation may be valid but the SLEE cannot perform the requested operation due to a system-level failure, or the

SLEE cannot determine whether the requested operation is valid due to a system-level failure.

- `java.lang.NullPointerException`.
  If any of a method's arguments are `null`, the method throws a `NullPointerException`.

14.14 Alarm Facility management interface

Changed in 1.1: The Alarm Facility now supports stateful alarms originating from a number of different sources in the SLEE. New methods have been added to the `AlarmManagementMBean` interface to support this new feature. The JMX Object Name of the `AlarmMBean` is now also specified.

The `AlarmMBean` interface defines the management interface of the Alarm Facility (see Section 13.2). An `AlarmMBean` object broadcasts alarm notifications for alarms raised and cleared in the SLEE. Alarms may originate from a number of sources including SBBs running in the SLEE, resource adaptor entities, profiles, and SLEE internal subsystems.

The `AlarmNotification` class defines the information conveyed in each alarm notification.

The base notification type for alarm notifications is "javax.slee.management.alarm". Each notification source in the SLEE that may generate alarms defines a notification type that is a subtype of the base alarm notification type. For example, the notification type for an alarm notification generated as a result of an SBB raising an alarm is "javax.slee.management.alarm.sbb". See Section 14.16 for a complete list of the notification sources defined by the SLEE specification and the alarm notification type used by each notification source.

The concrete class that implements the `AlarmMBean` interface must extend the `NotificationBroadcaster` interface even though the `AlarmMBean` interface does not extend the `NotificationBroadcaster` interface (see Section 14.3.1).

A management client can use a notification filter to filter alarm notifications. The SLEE specification defines a number of standard alarm notification filters:

- `AlarmLevelFilter`.
  This filter only allows alarms of a certain severity or higher to pass to the notification listener.
- `AlarmDuplicateFilter`.
  This filter suppresses equivalent alarm notifications that occur during a specified time period. Only the first in the series of duplicates are passed on to the notification listener. The cycle restarts when the time period expires.
- `AlarmThresholdFilter`.
  This filter suppresses equivalent alarm notifications until a specified number have occurred within a specified time period. If, within the same time period, the filter receives a further equivalent alarm notification, the notification is passed on to the notification listener and the cycle restarts. The cycle also restarts when the time period from the first notification expires without a notification being sent.

The Object Name of an `AlarmMBean` object is defined by the string "`javax.slee.management:name=Alarm`". *(Added in 1.1)*

The Object Name may also be obtained by invoking the `getAlarmMBean` method on a `SleeManagementMBean` object (see Section 14.5).

14.14.1 `AlarmMBean` interface

The `AlarmMBean` interface is as follows:

```
package javax.slee.management;
```

```
import javax.slee.UnrecognizedAlarmException;

public interface AlarmMBean {
        // JMX Object Name string of the AlarmMBean
        public static final String OBJECT_NAME = "javax.slee.management:name=Alarm";

        // methods
        public String[] getAlarms()
                throws  ManagementException;
        public String[] getAlarms(NotificationSource notificationSource)
                throws  NullPointerException,
                        UnrecognizedNotificationSourceException,
                        ManagementException;
        public Alarm getDescriptor(String alarmId)
                throws NullPointerException,
                        ManagementException;
        public Alarm getDescriptors(String[] alarmIds)
                throws NullPointerException,
                        ManagementException;
        public boolean isActive(String alarmID)
                throws  NullPointerException,
                        ManagementException;
        public boolean clearAlarm(String alarmID)
                throws  NullPointerException,
                        ManagementException;
        public int clearAlarms(NotificationSource notificationSource)
                throws  NullPointerException,
                        UnrecognizedNotificationSourceException,
                        ManagementException;
        public int clearAlarms(NotificationSource notificationSource, String alarmType)
                throws  NullPointerException,
                        UnrecognizedNotificationSourceException,
                        ManagementException;
}
```

- The `OBJECT_NAME` constant. *(Added in 1.1)*
  This constant defines the string form of the `AlarmMBean`'s JMX Object Name.
- The `getAlarms()` method. *(Added in 1.1)*
  This method returns the identifiers of the active alarms in the SLEE. If no alarms are active in the SLEE this method returns an empty array.
- The `getAlarms(NotificationSource)` method. *(Added in 1.1)*
  This method returns the identifiers of active alarms raised by the specified notification source. If no alarms are active this method returns an empty array.
  This method will throw an `UnrecognizedNotificationSourceException` if the `notificationSource` argument does not represent a notification source recognizable by the SLEE.
- The `isActive` method. *(Added in 1.1)*
  This method determines if the alarm specified by the `alarmID` parameter is active in the SLEE. This method can be used to determine if an alarm raised in the past is still active.
- The `getDescriptor` method. *(Added in 1.1)*
  This method returns the `Alarm` object for the alarm with the identifier specified by the `alarmID` argument. If the specified alarm is not known by the SLEE, this method returns `null`, allowing a management client to determine whether an alarm is still active, and obtain the information for the alarm if so, in one method call.
- The `getDescriptors` method. *(Added in 1.1)*
  This method performs the function of the above `getDescriptor` method over an array of alarm identifiers. The returned array has the same length as the array passed in by the `alarmsIDs` argument, and if `alarms = getDescriptors(ids)`, then `alarms[i] == getA-`

`larm(alarmIDs[i])`. If the `alarmsIDs` array contains any alarm identifier that is unrecognized by the SLEE, the returned array contains a `null` value at the corresponding array index position.

- The `clearAlarm` method. *(Added in 1.1)*
  This method clears the alarm with the identifier specified by the `alarmID` argument. If the specified alarm does not exist this method performs no function and returns `false`. Otherwise it clears the alarm and returns `true`.
- The `clearAlarms(NotificationSource)` method. *(Added in 1.1)*
  This method clears all active alarms raised by the specified notification source. If no active alarms exist for the notification source, this method has no effect. This method returns the number of alarms that were cleared.
  This method will throw an `UnrecognizedNotificationSourceException` if the `notificationSource` argument does not represent a notification source recognizable by the SLEE.
- The `clearAlarms(NotificationSource, String)` method. *(Added in 1.1)*
  This method clears all active alarms of the specified type raised by the specified notification source. If no active alarms of the specified type are found, this method has no effect. This method returns the number of alarms that were cleared.
  This method will throw an `UnrecognizedNotificationSourceException` if the `notificationSource` argument does not represent a notification source recognizable by the SLEE.
- The above methods may throw the following exceptions:
  - `javax.slee.management.ManagementException`.
    If the requested operation cannot be performed due to a system-level failure, then the method throws a `ManagementException`. The requested operation may be valid but the SLEE cannot perform the requested operation due to a system-level failure, or the SLEE cannot determine whether the requested operation is valid due to a system-level failure.
  - `java.lang.NullPointerException`.
    If any of a method's arguments are `null`, the method throws a `NullPointerException`.

14.14.2 `Alarm` class

Added in 1.1.

The `Alarm` class contains information about an alarm raised in the SLEE.

The public interface of the `Alarm` class is as follows:

```
package javax.slee.management;

import java.io.Serializable;

public final class Alarm implements VendorExtensions, Serializable, Cloneable, Comparable
{
        // constructors
        public Alarm(String alarmId, NotificationSource notificationSource,
                     String alarmType, String instanceId, AlarmLevel level,
                     String message, Throwable cause, long timestamp) { … }

        // accessors
        public String getAlarmID() { … }
        public NotificationSource getNotificationSource() { … }
        public String getAlarmType() { … }
```

```
        public String getInstanceID() { … }
        public AlarmLevel getAlarmLevel() { … }
        public String getMessage() { … }
        public Throwable getCause() { … }
        public long getTimestamp() { … }


        // vendor-specific extension data support
        public static void enableVendorDataSerialization() { … }
        public static void disableVendorDataSerialization() { … }
        public static void enableVendorDataDeserialization() { … }
        public static void disableVendorDataDeserialization() { … }
        public final void setVendorData(Object vendorData) { … }
        public final Object getVendorData() { … }

        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
        public int compareTo(Object obj) { … }
        public Object clone() { … }
}
```

- The constructor creates an `Alarm` object. It takes the following arguments:
  - The `alarmId` argument specifies the SLEE generated identifer of a raised alarm. For more information refer to section 13.2.1.
  - The `notificationSource` argument specifies the component which raised the alarm.
  - The `alarmType` argument specifies the type of the alarm. The type of an alarm is specified when the alarm is raised.
  - The `instanceId` argument specifies the instance of the alarm type. This is specified when the alarm is raised.
  - The `alarmLevel` argument specifies the severity of the alarm.
  - The `message` argument specifies the message describing the alarm.
  - The `cause` argument specifies an optional cause for the alarm. The cause is typically the exception that caused this alarm (if any). The value of this argument should be `null` if no cause was provided when the alarm was raised.
  - The `timestamp` argument specifies the timestamp when the alarm was raised. The timestamp is specified as the number of milliseconds since January 1, 1970 00:00:00 GMT (same as `Date.getTime()` and `System.currentTimeMillis()`).
- The `getAlarmID` method.
  This method returns the `alarmId` argument to the constructor.
- The `getNotificationSource` method.
  This method returns the `notificationSource` argument to the constructor.
- The `getAlarmType` method.
  This method returns the `alarmType` argument to the constructor.
- The `getInstanceID` method.
  This method returns the `instanceId` argument to the constructor.
- The `getAlarmLevel` method.
  This method returns the `alarmLevel` argument to the constructor.
- The `getMessage` method.
  This method returns the `message` argument to the constructor.

- The `getCause` method.
  This method returns the `cause` argument to the constructor.
- The `getTimestamp` method.
  This method returns the `timestamp` argument to the constructor.
- The `enableVendorDataSerialization`, `disableVendorDataSerialization`, `enableVendorDataDeserialization`, `disableVendorDataDeserialization`, `setVendorData`, and `getVendorData` methods.
  These methods provide support for vendor-specific data in `Alarm` objects. See Section 14.17 for details and an explanation of these methods.
- The `equals` method.
  This method indicates whether the object specified by the `obj` argument is equal to this alarm object. The specified object is equal to this alarm if the specified object is an `Alarm` object with the same alarm identifier, alarm type, and alarm history as this alarm.
- The `hashCode` method.
  This method provides a hash code value for the alarm.
- The `toString` method.
  This method provides a string representation for the alarm.
- The `compareTo` method.
  This method implements the java.lang.`Comparable` interface allowing alarms to be stored in ordered collections.
- The `clone` method.
  This method creates a copy of the alarm object.

14.14.3 `AlarmNotification` class

Changed in 1.1 to reflect changes to the Alarm MBean, and allow the source of the alarm to be any notification source defined in 1.1.

Alarm notifications are emitted by an Alarm MBean to indicate that some component or subsystem is experiencing a problem.

The public interface of the `AlarmNotification` class is as follows:

```
package javax.slee.management;

import javax.slee.management.Notification;
import javax.slee.facilities.AlarmLevel;
import javax.slee.facilities.Level;

public final class AlarmNotification extends Notification implements VendorExtensions {
        // constructor
        public AlarmNotification(String type, AlarmMBean alarmMBean,
                String alarmID, NotificationSource notificationSource,
                String alarmType, String instanceId, AlarmLevel alarmLevel,
                String message, Throwable cause, long sequenceNumber,
                long timestamp) { … }

        // deprecated constructor
        public AlarmNotification(AlarmMBean alarmMBean, String alarmType,
                Object alarmSource, Level alarmLevel, String message, Throwable cause,
                long sequenceNumber, long timestamp) { … }

        // accessors
        public String getAlarmID() { … }
        public NotificationSource getNotificationSource() { … }
        public String getAlarmType() { … }
        public String getInstanceID() { … }
```

```
        public AlarmLevel getAlarmLevel() { … }
        public Throwable getCause() { … }

        // vendor-specific extension data support
        public static void enableVendorDataSerialization() { … }
        public static void disableVendorDataSerialization() { … }
        public static void enableVendorDataDeserialization() { … }
        public static void disableVendorDataDeserialization() { … }
        public void setVendorData(Object vendorData) { … }
        public Object getVendorData() { … }

        // deprecated methods
        public Object getAlarmSource() { … }
        public Level getLevel() { … }

        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
}
```

The `AlarmNotification` class inherits all the attributes of the base `Notification` class. Of the inherited attributes, the following attributes have specific meaning defined by the SLEE specification:

- The `type` attribute.
  This attribute specifies the type of the alarm notification being generated. The type of the notification can be used to infer the type of the `NotificationSource` attribute of the alarm notification.
- The `source` attribute.
  This attribute specifies the `AlamMBean` that emitted this notification.

The `AlarmNotification` class adds the following additional attributes specific to alarm notifications:

- The `alarmID` attribute. *(Added in 1.1)*
  This attribute identifies the alarm that this alarm notification carries information about.
- The `notificationSource` attribute. *(Added in 1.1)*
  This attribute provides additional information about the source of the alarm. The extact type of this attribute should be able to be inferred from the `type` attribute of the alarm notification. For example, if the type is equal to `SbbNotification.ALARM_NOTIFICATION_TYPE` then the `notificationSource` attribute can be cast to an `SbbNotification` object to obtain the component identifiers of the service and SBB that caused the alarm notification to be generted.
- The `alarmType` attribute.
  This attribute specifies the alarm type of the alarm being generated. The alarm type is set by the alarm source.
- The `instanceId` attribute. *(Added in 1.1)*
  This attribute specifies the instance of the alarm being generated. The `instanceId` is set by the alarm source.
- The `alarmLevel` attribute. *(Added in 1.1)*
  This attribute specifies the alarm level of the alarm represented by the alarm notification.
- The `message` attribute.
  This attribute specifies the message of the alarm notification.
- The `cause` attribute.
  This attribute is typically used to propagate an exception in the alarm notification. The value of this attribute is `null` if the source of the alarm notification did not provide a value.
- The `alarmSource` attribute. *(Deprecated in 1.1)*
  This attribute was defined in the SLEE 1.0 version of this class. It is deprecated in SLEE 1.1 as it

was considered too generic to identify the source of an alarm. The `notificationSource` attribute is now used to identify the source of an alarm notification.

- The `level` attributes. *(Deprecated in 1.1)*
  This attribute was defined in the SLEE 1.0 version of this class. It is deprecated in SLEE 1.1 as industry standard alarm levels are now specified by the `AlarmLevel` class. The `alarmLevel` attribute now identifies the alarm level of an alarm notification.

The following methods are defined by the `AlarmNotification` class:

- The SLEE 1.1-defined constructor. *(Added in 1.1)*
  This constructor takes the following arguments:
  - The `type` argument specifies the value of the inherited `type` attribute for the notification.
  - The `alarmMBean` argument specifies the value of the inherited `source` attribute for the notification.
  - The `alarmID` argument specifies the value of the `alarmID` attribute for the notification.
  - The `notificationSource` argument specifies the value of the `notificationSource` attribute for the notification.
  - The `alarmType` argument specifies the value of the `alarmType` attribute for the notification.
  - The `instanceId` argument specifies the value of the `instanceId` attribute for the notification.
  - The `alarmLevel` argument specifies the value of the `alarmLevel` attribute for the notification.
  - The `message` argument specifies the value of the `message` attribute for the notification.
  - The `cause` argument specifies the value of the `cause` attribute for the notification.
  - The `sequenceNumber` argument specifies the value of the inherited `sequenceNumber` attribute for the notification.
  - The `timestamp` argument specifies the value of the inherited `timestamp` attribute for the notification.
- The SLEE 1.0-defined constructor. *(Deprecated in 1.1)*
  This constructor takes the following arguments:
  - The `alarmMBean` argument specifies the value of the inherited `source` attribute for the notification.
  - The `alarmType` argument specifies the value of the `alarmType` attribute for the notification.
  - The `alarmSource` argument specifies the value of the deprecated `alarmSource` attribute for the notification.
  - The `alarmLevel` argument specifies the value of the deprecated `level` attribute for the notification.
  - The `message` argument specifies the value of the `message` attribute for the notification.
  - The `cause` argument specifies the value of the `cause` attribute for the notification.

  - The `sequenceNumber` argument specifies the value of the inherited `sequenceNumber` attribute for the notification.
  - The `timestamp` argument specifies the value of the inherited `timestamp` attribute for the notification.

- The `getAlarmID` method. *(Added in 1.1)*
  This method returns the value of the `alarmID` attribute.
- The `getNotificationSource` method. *(Added in 1.1)*
  This method returns the value of the `notificationSource` attribute.
- The `getAlarmType` method.
  This method returns the value of the `alarmType` attribute.
- The `getInstanceID` method. *(Added in 1.1)*
  This method returns the value of the `instanceId` attribute.
- The `getAlarmLevel` method. *(Added in 1.1)*
  This method returns the value of the `alarmLevel` attribute.
- The `getCause` method.
  This method returns the value of the `cause` attribute.
- The `getAlarmSource` method. *(Deprecated in 1.1)*
  This method returns the value of the deprecated `alarmSource` attribute. For `AlarmNotification` objects created using the SLEE 1.1-defined constructor, this method returns the value of the `notificationSource` attribute.
- The `getLevel` method. *(Deprecated in 1.1)*
  This method returns the value of the deprecated `level` attribute. For `AlarmNotification` objects created using the SLEE 1.1-defined constructor, this method returns `javax.slee.facilities.Level.INFO`.
- The `enableVendorDataSerialization`, `disableVendorDataSerialization`, `enableVendorDataDeserialization`, `disableVendorDataDeserialization`, `setVendorData`, and `getVendorData` methods. *(Added in 1.1)*
  These methods provide support for vendor-specific data in `AlarmNotification` objects. See Section 14.17 for details and an explanation of these methods.
- The `equals` method.
  Two `AlarmNotification` objects are equivalent, i.e. `alarmOne.equals(alarmTwo)`, if their `alarmId`, `notificationSource`, `alarmType`, `instanceId`, `alarmLevel`, and `message` attributes are equal, as determined by their corresponding `equals` methods. Two `AlarmNotification` objects each created using the deprecated constructor are equivalent if their `alarmSource`, `alarmType`, `level`, and `message` attributes are equal, as determined by their corresponding `equals` methods. Two `AlarmNotification` object created using different constructors will never be equal.
- The `hashCode` method.
  This method returns the hash code of the `message` attribute.
- The `toString` method.
  This method provides a string representation for the alarm notification.

The object that raises or clears an alarm (the notification source) provides the values for the alarm notification's `alarmType`, `instanceId`, `alarmLevel`, `message`, and `cause` attributes. The `AlarmMBean` object that generates the alarm notification provides the values for the `type`, `source`, `alarmID`, `notificationSource`, `sequenceNumber` and `timestamp` attributes. (The `AlarmNotification`

class inherits the `type`, `source`, `message`, `sequenceNumber`, and `timestamp` attributes from the base `Notification` class.)

If an SBB uses the Alarm Facility to generate an alarm notification using one of the deprecated SLEE 1.0 methods, an `AlarmNotification` object must be created using the deprecated `AlarmNotification` constructor. In this case, the SBB provides the values for the alarm notification's `alarmSource`, `alarmType`, `level`, `message`, `cause`, and `timestamp` attributes. The `AlarmMBean` object that generates the alarm notification provides the values for the `source` and `sequenceNumber` attributes.

14.14.4 `AlarmLevelFilter` class

Changed in 1.1: The `AlarmLevelFilter` *has been changed to support all notification sources defined in 1.1, and filters using both the new* `AlarmLevel` *class as well as the deprecated* `Level` *class.*

The `AlarmLevelFilter` class filters alarm notifications based on the severity of the alarm level in the notification. Only alarm notifications of the specified severity level or greater will be allowed through this filter. There are two variants of this filter as described below:

- The first variant uses the `AlarmLevel` class introduced in the SLEE 1.1 specification (see Section 13.2.3) to compare alarm levels. An alarm notification may pass an `AlarmLevelFilter` of this variant if the `minLevel` argument to the constructor of this filter is less than or equal to the `alarmLevel` attribute of the alarm notification. More precisely, an alarm notification, `t`, will pass through this filter if `!minLevel.isHigherLevel(t.getAlarmLevel())`, where `minLevel` is the constructor parameter for the `AlarmLevelFilter`.
- The second variant is the version originally defined in the SLEE 1.0 specification, and is deprecated in the SLEE 1.1 specification as it uses the deprecated `Level` class (see Section 13.4) to compare alarm levels. An alarm notification may pass an `AlarmLevelFilter` of this variant if the `minLevel` argument to the constructor of this filter is less than or equal to the `level` attribute of the alarm notification. More precisely, an alarm notification, `t`, will pass through this filter if `!minLevel.isHigherLevel(t.getLevel())`, where `minLevel` is the constructor parameter for the `AlarmLevelFilter`.

The public interface of the `AlarmLevelFilter` class is as follows:

```
package javax.slee.management;

import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.slee.facilities.Level;
import javax.slee.facilities.AlarmLevel;

public class AlarmLevelFilter implements NotificationFilter {
        // constructor
        public AlarmLevelFilter(AlarmLevel minLevel) { … }

        // deprecated constructor
        public AlarmLevelFilter(Level minLevel) { … }

        // methods
        public boolean isNotificationEnabled(Notification notification) { … }
}
```

- The SLEE 1.1-defined constructor. *(Added in 1.1)*
  This constructor must be used to filter `AlarmNotification` objects created using the SLEE 1.1-defined `AlarmNotification` class constructor (see Section 14.14.3). An `AlarmLevelFilter` object created with this constructor will completely suppress alarm notifications whose `AlarmNotification` object was created using the deprecated SLEE 1.0-defined `AlarmNotification` class constructor.
  This constructor takes the following arguments:

  - The `minLevel` argument specifies minimum alarm level of alarm notifications allowed through this filter.

- The SLEE 1.0-defined constructor. *(Deprecated in 1.1)*
  This constructor must be used to filter `AlarmNotification` objects created using the deprecated SLEE 1.0-defined `AlarmNotification` class constructor (see Section 14.14.3). An `AlarmLevelFilter` object created with this constructor will completely suppress alarm notifications whose `AlarmNotification` object was created using the SLEE 1.1-defined `AlarmNotification` class constructor.
  This constructor takes the following arguments:
  - The `minLevel` argument specifies minimum alarm level of alarm notifications allowed through this filter.
- The `isNotificationEnabled` method.
  This method implements the `NotificationFilter` interface method to determine if the `notification` argument satisfies the filter criteria.

14.14.5 `AlarmDuplicateFilter` class

The `AlarmDuplicateFilter` class suppresses equivalent alarm notifications that occur during a specified time period. Only the first in a series of equivalent alarm notifications is allowed through this filter. When the specified period of time elapses after the first notification, the cycle restarts and a subsequent equivalent alarm notification is allowed through. This filter can be used to suppress repetitive alarms that occur frequently.

As an example, consider an `AlarmDuplicateFilter` object with a period set to 5000ms. On observation of an alarm notification that is not equivalent to any others it has seen, the filter notes the current time, remembers the notification, and passes the notification on to the notification listener. If further equivalent alarm notifications are observed by the filter within the 5000ms time period, they are suppressed. Once the 5000ms time period elapses, the original notification is forgotten and the cycle restarts.

The public interface of the `AlarmDuplicateFilter` class is as follows:

```
package javax.slee.management;

import javax.management.Notification;
import javax.management.NotificationFilter;

public class AlarmDuplicateFilter implements NotificationFilter {
       // constructor
       public AlarmDuplicateFilter(long period) { … }

       // methods
       public boolean isNotificationEnabled(Notification notification) { … }
}
```

- The constructor.
  The constructor takes the following argument:
  - The `period` argument specifies the time period, in milliseconds, in which duplicate alarms will be suppressed.
- The `isNotificationEnabled` method.
  This method implements the `NotificationFilter` interface method to determine if the `notification` argument satisfies the filter criteria.

14.14.6 `AlarmThresholdFilter` class

The `AlarmThresholdFilter` class filters equivalent alarm notifications until a specified number have occurred within a specified time period. Once the threshold has been reached, the next equivalent alarm

notification is allowed through this filter and the cycle restarts. The cycle also restarts if the time period from the first notification expires without the threshold being passed. This filter can be used to suppress alarms that occur and then clear up spontaneously.

As an example, consider an `AlarmThresholdFilter` object with a threshold of five notifications and a timeout of 500ms. On observation of an alarm notification that is not equivalent to any others it has seen, the filter notes the current time and remembers the notification. If a further five equivalent alarm notifications are observed by the filter within the 500ms timeout period, the last notification is passed on to the notification listener and the original notification is forgotten. If the 500ms time period expires without the five equivalent notifications being observed, the original notification is forgotten and the cycle restarts.

The public interface of the `AlarmThresholdFilter` class is as follows:

```
package javax.slee.management;

import javax.management.Notification;
import javax.management.NotificationFilter;

public class AlarmThresholdFilter implements NotificationFilter {
       // constructor
       public AlarmThresholdFilter(int threshold, long timeout) { … }

       // methods
       public boolean isNotificationEnabled(Notification notification) { … }
}
```

- The constructor.
  The constructor takes the following arguments:
  - The `threshold` argument specifies the number of duplicate alarms that must be seen before the next duplicate alarm notification is allowed through this filter.
  - The `timeout` argument specifies the timeout period, in milliseconds, after which the count of duplicates previously seen for a particular alarm notification will be reset if the threshold limit has not been met.
- The `isNotificationEnabled` method.
  This method implements the `NotificationFilter` interface method to determine if the `notification` argument satisfies the filter criteria.

14.15 Trace Facility management interface

Changed in 1.1: The `TraceFacility` interface has been deprecated and replaced with `Tracer` objects. New methods have been added to the `TraceMBean` interface to support this new feature. The JMX Object Name of the `AlarmMBean` is now also specified.

The `TraceMBean` interface defines the management interface of the Trace Facility (see Section 13.3). A `TraceMBean` object broadcasts trace notifications for trace messages emitted by the various notification sources in the SLEE. The `TraceNotification` class defines the information conveyed in each trace notification.

The `TraceMBean` interface is an MBean interface that defines methods to discover the names of the tracers that are being used by notification sources in the SLEE and to get and set the trace filter levels for those tracers. The notification sources that trace notifications may originate from include SBBs running in the SLEE, resource adaptor entities, Profiles, and SLEE internal subsystems.

The `TraceNotification` class defines the information conveyed in each trace notification. The base notification type for trace notifications is “javax.slee.management.trace”. Each notification source in the SLEE that may generate trace messages defines a notification type that is a subtype of the base trace notification type. For example, the notification type for a trace notification generated as a result of an SBB emitting a trace message is “javax.slee.management.trace.sbb”. See Section 14.16 for a complete list of the noti-

fication sources defined by the SLEE specification and the trace notification type used by each notification source.

The concrete class that implements the `TraceMBean` interface must extend the `NotificationBroadcaster` interface even though the `TraceMBean` interface does not extend the `NotificationBroadcaster` interface (see Section 14.3.1).

A management client can use a notification filter to filter trace notifications. The SLEE specification defines one standard trace notification filter:

- `TraceLevelFilter`.
  This filter only allows trace notifications of a certain trace level or higher to pass to the notification listener.

The Object Name of a `TraceMBean` object is defined by the string "`javax.slee.management:name=Trace`". *(Added in 1.1)*

The Object Name may also be obtained invoking the `getTraceMBean` method on a `SleeProviderMBean` object (see Section 14.5).

14.15.1 `TraceMBean` interface

The `TraceMBean` interface is as follows:

```
package javax.slee.management;

import javax.slee.ComponentID;
import javax.slee.InvalidArgumentException;
import javax.slee.facilities.Level;
import javax.slee.facilities.TaceLevel;

public interface TraceMBean {
        // JMX Object Name string of the TraceMBean
        public static final String OBJECT_NAME = "javax.slee.management:name=Trace";

        // methods
        public String[] getTracersUsed(NotificationSource notificationSource)
                throws  NullPointerException,
                        UnrecognizedNotificationSourceException,
                        ManagementException;
        public void setTraceLevel(NotificationSource notificationSource,
                                String tracerName, TraceLevel level)
                throws  NullPointerException,
                        UnrecognizedNotificationSourceException,
                        ManagementException;
        public void unsetTraceLevel(NotificationSource notificationSource,
                                String tracerName)
                throws  NullPointerException,
                        UnrecognizedNotificationSourceException,
                        InvalidArgumentException,
                        ManagementException;
        public void getTraceLevel(NotificationSource notificationSource,
                                String tracerName)
                throws  NullPointerException,
                        InvalidArgumentException,
                        UnrecognizedNotificationSourceException,
                        ManagementException;
        public String[] getTracersSet(NotificationSource notificationSource)
                throws  NullPointerException,
                        UnrecognizedNotificationSourceException,
                        ManagementException;

        // deprecated
        public void setTraceLevel(ComponentID id, Level traceLevel)
                throws  NullPointerException,
                        UnrecognizedComponentException,
```

```
                    ManagementException;
        public Level getTraceLevel(ComponentID id)
               throws  NullPointerException,
                       UnrecognizedComponentException,
                       ManagementException;
}
```

- The `OBJECT_NAME` constant. *(Added in 1.1)*
  This constant defines the string form of the `TraceMBean`'s JMX Object Name.
- The `getTracersUsed` method. *(Added in 1.1)*
  This method returns the list of tracer names for which `Tracer` objects have been requested by the notification source identified by the `notificationSource` argument. If no tracers have been requested by the notification source this method returns an empty array.
- The `setTraceLevel(NotificationSource, String, TraceLevel)` method. *(Added in 1.1)*
  This method sets the trace filter level of the tracer indentified by the `tracerName` argument to the trace level specified by the `traceLevel` argument for the notification source identified by the `notificationSource` argument. A trace message emitted by a notification source to a tracer at an equal or higher trace level than that effective trace filter level set for that tracer causes a trace notification to be generated by the `TraceMBean` object. Conversely, a trace message emitted by a notification source to a tracer at a trace level that is less than the trace filter level set for that tracer does not cause a trace notification to be generated. If the trace filter level for a tracer is set to `TraceLevel.OFF` (see Section 13.3.2), then no trace notifications are generated for trace messages emitted to that tracer.

  The SLEE specification recommends that the trace filter levels set for tracers and notification source be persistent across SLEE restarts, but does not mandate this as a requirement.
- The `unsetTraceLevel` methods. *(Added in 1.1)*
  This method removes the trace filter level set for the tracer identified by the `tracerName` argument for a the notification source identified by the `notificationSource` argument. This method has no effect, and returns without error, if no trace filter level is currently set for the specified tracer. The trace filter level for a root tracer (identified by the empty string) cannot be unset. This method throws the following exception:
  - `javax.slee.InvalidArgumentException`.
    This exception is thrown if the tracer name passed in as an argument is an empty string, identifying the root tracer.
- The `getTraceLevel(NotificationSource, String)` method.*(Added in 1.1)*
  This method returns the effective trace filter level of the tracer identified by the `tracerName` attribute for the notification source identified by the `notificationSource` argument. Note that the returned trace level may not be directly set for the specified tracer, but may have been obtained from the tracer's nearest ancestor that has a trace filter level set. This method throws the following exception:
  - `javax.slee.InvalidArgumentException`.
    This exception is thrown if the tracer name passed is not a valid tracer name.
- The `getTracersSet` methods. *(Added in 1.1)*
  This method returns the list of tracer names for which a trace filter level has been set for the notification source identified by the `notificationSource` argument.
- The `setTraceLevel(ComponentID, Level)` and `getTraceLevel(ComponentID)` methods. *(Deprecated in 1.1)*
  These methods have been deprecated in SLEE 1.1 as they associate trace levels only with singular component identifiers corresponding to components installed in the SLEE. This was both ambigu-

ous with respect to SBB tracing (e.g. does the SBB emit trace message using its SBB identifier or Service identifier?) and insufficient in scope to handle logical entities created from components in the SLEE, such as resource adaptor entities.
These methods may throw the following exception:

- `javax.slee.UnrecognizedComponentException`.
  This exception is thrown if the component identifier passed in as an argument is not recognized by the SLEE as a valid identifier.

- The above methods may throw the following exceptions:
  - `javax.slee.UnrecognizedNotificationSourceException`.
    This exception is thrown if the `notificationSource` argument passed to the method is not recognized by the SLEE as a valid notification source.
  - `javax.slee.management.ManagementException`.
    If the requested operation cannot be performed due to a system-level failure, then the method throws a `ManagementException`. The requested operation may be valid but the SLEE cannot perform the requested operation due to a system-level failure, or the SLEE cannot determine whether the requested operation is valid due to a system-level failure.
  - `java.lang.NullPointerException`.
    If any of a method's arguments are `null`, the method throws a `NullPointerException`.
- The default trace filter level for all tracers is `TraceLevel.INFO`. Thus the default effective trace filter level for all tracers is also `TraceLevel.INFO`.

14.15.2 `TraceNotification` class

Changed in 1.1 to reflect changes to the Trace MBean, and allow the source of trace messages to be any notification source defined in 1.1.

Trace notifications are emitted by a `TraceMBean` when a notification source emits a trace message using a `Tracer` object at a trace level equal to or higher than the effective trace filter level set for the tracer. Trace filter levels may be set for tracers used by a notification source using the methods on the `TraceMBean` interface.

The public interface of the `TraceNotification` class is as follows:

```
package javax.slee.management;

import javax.management.Notification;
import javax.slee.facilities.TraceLevel;

public final class TraceNotification extends Notification implements VendorExtensions {
        // constructor
        public TraceNotification(String type, TraceMBean traceMBean,
                NotificationSource notificationSource, String tracerName,
                TraceLevel traceLevel, String message, Throwable cause,
                long sequenceNumber, long timestamp) { … }

        // deprecated constructor
        public TraceNotification(TraceMBean traceMBean, String messageType,
                Object messageSource, Level traceLevel, String message, Throwable cause,
                long sequenceNumber, long timestamp) { … }

        // accessors
        public NotificationSource getNotificationSource() { … }
        public String getTracerName() { … }
        public TraceLevel getTraceLevel() { … }
        public Throwable getCause() { … }
```

```
        // vendor-specific extension data support
        public static void enableVendorDataSerialization() { … }
        public static void disableVendorDataSerialization() { … }
        public static void enableVendorDataDeserialization() { … }
        public static void disableVendorDataDeserialization() { … }
        public final void setVendorData(Object vendorData) { … }
        public final Object getVendorData() { … }

        // deprecated methods
        public String getMessageType() { … }
        public Object getMessageSource() { … }
        public Level getLevel() { … }

        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
}
```

The `TraceNotification` class inherits all the attributes of the base `Notification` class. Of the inherited attributes, the following attributes have specific meaning defined by the SLEE specification:

- The `type` attribute.
  This attribute specifies the type of the trace notification being generated. The type of the notification can be used to infer the type of the `NotificationSource` attribute of the trace notification.
- The `source` attribute.
  This attribute specifies the `TraceMBean` that emitted this notification.

The `TraceNotification` class adds the following additional attributes specific to trace notifications:

- The `notificationSource` attribute. *(Added in 1.1)*
  This attribute provides additional information about the source of the trace message. The extact type of this attribute should be able to be inferred from the `type` attribute of the trace notification. For example, if the type is equal to `SbbNotification.TRACE_NOTIFICATION_TYPE` then the `notificationSource` attribute can be cast to an `SbbNotification` object to obtain the component identifiers of the service and SBB that generated the trace message.
- The `tracerName` attribute. *(Added in 1.1)*
  This attribute specifies the name of the tracer to which this trace was generated.
- The `traceLevel` attribute. *(Added in 1.1)*
  This attribute specifies the trace level of the trace message contained in the trace notification.
- The `message` attribute.
  This attribute specifies the message of the trace notification.
- The `cause` attribute.
  This attribute is typically used to propagate an exception in the trace notification. The value of this attribute is `null` if the source of the trace notification did not provide a value.
- The `messageType` attribute. *(Deprecated in 1.1)*
  This attribute was defined in the SLEE 1.0 version of this class. It is deprecated in SLEE 1.1 and is replaced by the `tracerName` attribute. The `messageType` and `tracerName` attributes effectively convey the same information.
- The `messageSource` attribute. *(Deprecated in 1.1)*
  This attribute was defined in the SLEE 1.0 version of this class. It is deprecated in SLEE 1.1 as it was considered too generic to identify the source of a trace message. The `notification-Source` attribute is now used to identify the source of a trace notification.

- The `level` attributes. *(Deprecated in 1.1)*
  This attribute was defined in the SLEE 1.0 version of this class. It is deprecated in SLEE 1.1 as trace levels are now specified by the `TraceLevel` class. The `traceLevel` attribute now identifies the trace level of a trace notification.

The following methods are defined by the `TraceNotification` class:

- The SLEE 1.1-defined constructor. *(Added in 1.1)*
  This constructor takes the following arguments:
  - The `type` argument specifies the value of the inherited `type` attribute for the notification.
  - The `traceMBean` argument specifies the value of the inherited `source` attribute for the notification.
  - The `notificationSource` argument specifies the value of the `notificationSource` attribute for the notification.
  - The `tracerName` argument specifies the value of the `tracerName` attribute for the notification.
  - The `traceLevel` argument specifies the value of the `traceLevel` attribute for the notification.
  - The `message` argument specifies the value of the `message` attribute for the notification.
  - The `cause` argument specifies the value of the `cause` attribute for the notification.
  - The `sequenceNumber` argument specifies the value of the inherited `sequenceNumber` attribute for the notification.
  - The `timestamp` argument specifies the value of the inherited `timestamp` attribute for the notification.
- The SLEE 1.0-defined constructor. *(Deprecated in 1.1)*
  This constructor takes the following arguments:
  - The `traceMBean` argument specifies the value of the inherited `source` attribute for the notification.
  - The `messageType` argument specifies the value of the `messageType` attribute for the notification.
  - The `messageSource` argument specifies the value of the deprecated `messageSource` attribute for the notification.
  - The `traceLevel` argument specifies the value of the deprecated `level` attribute for the notification.
  - The `message` argument specifies the value of the `message` attribute for the notification.
  - The `cause` argument specifies the value of the `cause` attribute for the notification.
  - The `sequenceNumber` argument specifies the value of the inherited `sequenceNumber` attribute for the notification.
  - The `timestamp` argument specifies the value of the inherited `timestamp` attribute for the notification.
- The `getNotificationSource` method. *(Added in 1.1)*
  This method returns the value of the `notificationSource` attribute.

- The `getTracerName` method. *(Added in 1.1)*
  This method returns the value of the `tracerName` attribute.
- The `getTraceLevel` method. *(Added in 1.1)*
  This method returns the value of the `traceLevel` attribute.
- The `getMessage` method.
  This method returns the value of the `message` attribute.
- The `getCause` method.
  This method returns the value of the `cause` attribute.
- The `getMessageType` method. *(Deprecated in SLEE 1.1)*
  This method returns the value of the deprecated `messageType` attribute.
- The `getMessageSource` method. *(Deprecated in SLEE 1.1)*
  This method returns the value of the deprecated `messageSource` attribute. For `TraceNotification` objects created using the SLEE 1.1-defined constructor, this method returns the value of the `notificationSource` attribute.
- The `getLevel` method. *(Deprecated in SLEE 1.1)*
  This method returns the value of the deprecated `level` attribute. For `TraceNotification` objects created using the SLEE 1.1-defined constructor, this method returns the `javax.slee.facilities.Level` object equivalent to the trace level identified by the `traceLevel` attribute.
- The `enableVendorDataSerialization`, `disableVendorDataSerialization`, `enableVendorDataDeserialization`, `disableVendorDataDeserialization`, `setVendorData`, and `getVendorData` methods. *(Added in 1.1)*
  These methods provide support for vendor-specific data in `TraceNotification` objects. See Section 14.17 for details and an explanation of these methods.
- The `equals` method.
  Two `TraceNotification` objects are equivalent, i.e. `traceOne.equals(traceTwo)`, if their `notificationSource`, `tracerName`, `traceLevel`, and `message` attributes are equal, as determined by their corresponding `equals` methods. Two `TraceNotification` objects each created using the deprecated constructor are equivalent if their `messageSource`, `messageType`, `level`, and `message` attributes are equal, as determined by their corresponding `equals` methods. Two `TraceNotification` object created using different constructors will never be equal.
- The `hashCode` method.
  This method returns the hash code of the `message` attribute.
- The `toString` method.
  This method provides a string representation for the trace notification.

The object that emits a trace message (the notification source) provides the values for the trace notification's `traceLevel`, `message` and `cause` attributes. The `TraceMBean` object that generates the trace notification provides the values for the notification's `type`, `traceMBean`, `notificationSource`, `tracerName`, `sequenceNumber` and `timestamp` attributes. (The `TraceNotification` class inherits the `type`, `message`, `sequenceNumber`, and `timestamp` attributes from the base `Notification` class.)

If an SBB uses a (deprecated) `TraceFacility` object to generate a trace notification, a `TraceNotification` object must be created using the deprecated `TraceNotification` constructor. In this case, the SBB provides the values for the trace notification's `messageSource`, `messageType`, `level`,

`message`, `cause`, and `timestamp` attributes. The `TraceMBean` object that generates the trace notification provides the values for the `source` and `sequenceNumber` attributes.

14.15.3 `TraceLevelFilter` class

Changed in 1.1: The `TraceLevelFilter` *has been changed to support all notification sources defined in 1.1, and filters using both the new* `TraceLevel` *class as well as the deprecated* `Level` *class.*

The `TraceLevelFilter` class filters trace notifications based on the trace level in each trace notification. Only trace notifications of the specified trace level or higher will be allowed through this filter. There are two variants of this filter as described below:

- The first variant uses the `TraceLevel` class introduced in the SLEE 1.1 specification (see Section 13.3.2) to compare trace levels. A trace notification may pass a `TraceLevelFilter` of this variant if the `minLevel` argument to the constructor of this filter is less than or equal to the `traceLevel` attribute of the trace notification. More precisely, a trace notification, `t`, will pass through this filter if `!minLevel.isHigherLevel(t.getTraceLevel())`, where `minLevel` is the constructor parameter for the `TraceLevelFilter`.
- The second variant is the version originally defined in the SLEE 1.0 specification, and is deprecated in the SLEE 1.1 specification as it uses the deprecated `Level` class (see Section 13.4) to compare trace levels. A trace notification may pass a `TraceLevelFilter` of this variant if the `minLevel` argument to the constructor of this filter is less than or equal to the `level` attribute of the trace notification. More precisely, a trace notification, `t`, will pass through this filter if `!minLevel.isHigherLevel(t.getLevel())`, where `minLevel` is the constructor parameter for the `TraceLevelFilter`.

The public interface of the `TraceLevelFilter` class is as follows:

```
package javax.slee.management;

import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.slee.facilities.Level;
import javax.slee.facilities.TraceLevel;

public class TraceLevelFilter implements NotificationFilter {
        // constructor
        public TraceLevelFilter(TraceLevel minLevel) { … }

        // deprecated constructor
        public TraceLevelFilter(Level minLevel) { … }

        // methods
        public boolean isNotificationEnabled(Notification notification) { … }
}
```

- The SLEE 1.1-defined constructor. *(Added in 1.1)*
  This constructor must be used to filter `TraceNotification` objects created using the SLEE 1.1-defined `TraceNotification` class constructor (see Section 14.15.2). A `TraceLevelFilter` object created with this constructor will completely suppress trace notifications whose `TraceNotification` object was created using the deprecated SLEE 1.0-defined `TraceNotification` class constructor.
  This constructor takes the following arguments:
  - The `minLevel` argument specifies minimum trace level of trace notifications allowed through this filter.
- The SLEE 1.0-defined constructor. *(Deprecated in 1.1)*
  This constructor must be used to filter `TraceNotification` objects created using the depre-

cated SLEE 1.0-defined `TraceNotification` class constructor (see Section 14.15.2). A `TraceLevelFilter` object created with this constructor will completely suppress trace notifications whose `TraceNotification` object was created using the SLEE 1.1-defined `TraceNotification` class constructor.
This constructor takes the following arguments:

- The `minLevel` argument specifies minimum trace level of trace notifications allowed through this filter.

- The `isNotificationEnabled` method.
  This method implements the `NotificationFilter` interface method to determine if the `notification` argument satisfies the filter criteria.

14.16 Notification Sources

Added in 1.1.

A notification source is a component installed in the SLEE, or subsystem of the SLEE, that is capable of causing alarm, trace, or usage notifications to be generated and broadcast to their respective MBean notification listeners. The SLEE specification defines the following notification source interface and classes:

- The `NotificationSource` interface is the common base interface of all notification source classes.
- The `SbbNotification` class for identifying an SBB in a service as a notification source.
- The `ProfileTableNotification` class for identifying a profile table as a notification source.
- The `ResourceAdaptorEntityNotification` class for identifying a resource adaptor entity as a notification source.
- The `SubsystemNotification` class for identifying an arbitrary internal vendor-defined SLEE subsystem as a notification source.

14.16.1 `NotificationSource` interface

Added in 1.1.

The `NotificationSource` interface is the common interface implemented by all notification source classes.

The public interface of the `NotificationSource` interface is as follows:

```
package javax.slee.management;

import java.io.Serializable;

public interface NotificationSource extends implements Comparable, Serializable {
       // JMX notification type accessors
       public String getAlarmNotificationType();
       public String getTraceNotificationType();
       public String getUsageNotificationType();

       public boolean equals(Object obj);
       public int hashCode();
       public String toString();
       public int compareTo(Object obj);
}
```

The concrete implementation class of a `NotificationSource` object provides an appropriate implementation of these methods.

- The `getAlarmNotificationType` method.
  This method returns the JMX notification type of alarm notifications generated in response to the notification source interacting with the Alarm Facility. If the notification source has no defined interface to the Alarm Facility, this method returns `null`.
- The `getTraceNotificationType` method.
  This method returns the JMX notification type of trace notifications generated in response to the notification source interacting with the Trace Facility. If the notification source has no defined interface to the Trace Facility, this method returns `null`.
- The `getUsageNotificationType` method.
  This method returns the JMX notification type of usage notifications generated in response to the notification source interacting with its usage parameters. If the notification source has no defined interface to usage parameters, this method returns `null`.
- The `equals` method.
  This method indicates whether the specified object (specified by the `obj` argument) is equal to this notification source. The specified object is equal to this notification source if the specified object is a notification source of the same type and identifies the same source.
- The `hashCode` method.
  The implementation of this method should provide a reasonable hash code value for the notification source.
- The `toString` method.
  The implementation of this method should provide a reasonable string representation for the notification source.
- The `compareTo` method.
  This method, inherited from the java.`lang`.Comparable interface, allows notification sources to be stored in ordered collections. The implementation of this method should provide some natural ordering to notification source objects. It should also allow any object that implements the NotificationSource interface to be compared for order, allowing different type of notification sources to be stored in a single ordered collection.

14.16.2 `SbbNotification` class

Added in 1.1.

The `SbbNotification` class identifies an SBB in a service as the source of a notification.

The public interface of the `SbbNotification` class is as follows:

```
package javax.slee.management;

import javax.slee.SbbID;
import javax.slee.ServiceID;
import javax.management.ObjectName;

public final class SbbNotification implements NotificationSource {
       // JMX notification type constants
       public static final String ALARM_NOTIFICATION_TYPE =
              "javax.slee.management.alarm.sbb";
       public static final String TRACE_NOTIFICATION_TYPE =
              "javax.slee.management.trace.sbb";
       public static final String USAGE_NOTIFICATION_TYPE =
              "javax.slee.management.usage.sbb";

       // additional JMX Object Name attributes of a Usage MBean for an SBB
       public static final String SERVICE_NAME_KEY = "serviceName";
       public static final String SERVICE_VENDOR_KEY = "serviceVendor";
       public static final String SERVICE_VERSION_KEY = "serviceVersion";
       public static final String SBB_NAME_KEY = "sbbName";
```

```
        public static final String SBB_VENDOR_KEY = "sbbVendor";
        public static final String SBB_VERSION_KEY = "sbbVersion";

        // constructor
        public SbbNotification(ServiceID service, SbbID sbb) { … }

        // accessors
        public ServiceID getService() { … }
        public SbbID getSbb() { … }

        // JMX notification type accessors
        public String getAlarmNotificationType() { … }
        public String getTraceNotificationType() { … }
        public String getUsageNotificationType() { … }

        // JMX Usage MBean Object Name helper methods
        public static String getUsageMBeanProperties(ServiceID service, SbbID sbb) { … }
        public String getUsageMBeanProperties() { … }

        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
        public int compareTo(Object obj) { … }
}
```

- The `ALARM_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in alarm notifications generated by the Alarm MBean when an SBB raises or clears an alarm. This notification type identifies the `NotificationSource` object in the `AlarmNotification` object as being of the type `SbbNotification`.
- The `TRACE_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in trace notifications generated by the Trace MBean when an SBB emits a trace message. This notification type identifies the `NotificationSource` object in the `TraceNotification` object as being of the type `SbbNotification`.
- The `USAGE_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in usage notifications generated by a Usage MBean when an SBB updates a usage parameter. This notification type identifies the `NotificationSource` object in the `UsageNotification` object as being of the type `SbbNotification`.
- The `SERVICE_NAME_KEY`, `SERVICE_VENDOR_KEY`, `SERVICE_VERSION_KEY`, `SBB_NAME_KEY`, `SBB_VENDOR_KEY`, and `SBB_VERSION_KEY` constants.
  These constants define the names of attributes included in the Object Name of a Usage MBean for an SBB. These attributes are in addition to the standard attributes required for a Usage MBean's Object Name (refer Section 14.9.1).
- The constructor.
  The constructor takes the following arguments:
  - The `service` argument specifies a service identifier that identifies a Service installed in the SLEE.
  - The `sbb` argument specifies an SBB identifier that identifies an SBB installed in the SLEE that is participating in the Service identified by the `service` argument.
- The `getService` method.
  This method returns the `service` argument of the constructor.
- The `getSbb` method.
  This method returns the `sbb` argument of the constructor.

- The `getAlarmNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `ALARM_NOTIFICATION_TYPE` constant defined in this class.
- The `getTraceNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `TRACE_NOTIFICATION_TYPE` constant defined in this class.
- The `getUsageNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `USAGE_NOTIFICATION_TYPE` constant defined in this class.
- The `getUsageMBeanProperties` methods.
  These methods return a JMX Object Name property string that can be used as part of the complete Object Name of a Usage MBean for an SBB. The zero-argument form of this method returns a property string for the `SbbNotification` object that it is invoked on. The static two-argument form of this method returns a property string for the given `service` and `sbb` arguments.
- The `equals` method.
  This method indicates whether the object specified by the `obj` argument is equal to this notification source. The specified object is equal to this notification source if the specified object is an `SbbNotification` object encapsulating the same service and SBB component identifiers as this.
- The `hashCode` method.
  This method provides a hash code value for the notification source.
- The `toString` method.
  This method provides a string representation for the notification source.
- The `compareTo` method.
  This method compares the object specified by the `obj` argument with this notification source for order.

14.16.3 `ProfileTableNotification` class

Added in 1.1.

The `ProfileTableNotification` class identifies a profile table as the source of a notification.

The public interface of the `ProfileTableNotification` class is as follows:

```
package javax.slee.management;

public final class ProfileTableNotification implements NotificationSource {
        // constants
        public static final String ALARM_NOTIFICATION_TYPE =
                "javax.slee.management.alarm.profiletable";
        public static final String TRACE_NOTIFICATION_TYPE =
                "javax.slee.management.trace.profiletable";
        public static final String USAGE_NOTIFICATION_TYPE =
                "javax.slee.management.usage.profiletable ";

        // additional JMX Object Name attribute of a Usage MBean for a
        // Profile Table
        public static final String PROFILE_TABLE_NAME_KEY = "profileTableName";

        // constructor
        public ProfileTableNotification(String profileTableName) { … }

        // accessors
        public String getProfileTableName() { … }
```

```
        // JMX notification type accessors
        public String getAlarmNotificationType() { … }
        public String getTraceNotificationType() { … }
        public String getUsageNotificationType() { … }

        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
        public int compareTo(Object obj) { … }
}
```

- The `ALARM_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in alarm notifications generated by the Alarm MBean when a Profile raises or clears an alarm. This notification type identifies the `NotificationSource` object in the `AlarmNotification` object as being of the type `ProfileTableNotification`.
- The `TRACE_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in trace notifications generated by the Trace MBean when a Profile emits a trace message. This notification type identifies the `NotificationSource` object in the `TraceNotification` object as being of the type `ProfileTableNotification`.
- The `USAGE_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in usage notifications generated by a Usage MBean when a Profile updates a usage parameter. This notification type identifies the `NotificationSource` object in the `UsageNotification` object as being of the type `ProfileTableNotification`.
- The constructor.
  The constructor takes the following arguments:
  - The `profileTableName` argument specifies the name of a profile table in the SLEE.
- The `getProfileTableName` method.
  This method returns the `profileTableName` argument of the constructor.
- The `getAlarmNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `ALARM_NOTIFICATION_TYPE` constant defined in this class.
- The `getTraceNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `TRACE_NOTIFICATION_TYPE` constant defined in this class.
- The `getUsageNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `USAGE_NOTIFICATION_TYPE` constant defined in this class.
- The `equals` method.
  This method indicates whether the object specified by the `obj` argument is equal to this notification source. The specified object is equal to this notification source if the specified object is a `ProfileTableNotification` object encapsulating the same profile table name as this.
- The `hashCode` method.
  This method provides a hash code value for the notification source.
- The `toString` method.
  This method provides a string representation for the notification source.

- The `compareTo` method.
  This method compares the object specified by the `obj` argument with this notification source for order.

14.16.4 `ResourceAdaptorEntityNotification` class

Added in 1.1.

The `ResourceAdaptorEntityNotification` class identifies a resource adaptor entity as the source of a notification.

The public interface of the `ResourceAdaptorEntityNotification` class is as follows:

```
package javax.slee.management;

public final class ResourceAdaptorEntityNotification implements NotificationSource {
        // constants
        public static final String ALARM_NOTIFICATION_TYPE =
                "javax.slee.management.alarm.raentity";
        public static final String TRACE_NOTIFICATION_TYPE =
                "javax.slee.management.trace.raentity";
        public static final String USAGE_NOTIFICATION_TYPE =
                "javax.slee.management.usage.raentity";

        // additional JMX Object Name attribute of a Usage MBean for a
        // resource adaptor entity
        public static final String RESOURCE_ADAPTOR_ENTITY_NAME_KEY = "raEntityName";

        // constructor
        public ResourceAdaptorEntityNotification(String entityName) { … }

        // accessors
        public String getEntityName() { … }

        // JMX notification type accessors
        public String getAlarmNotificationType() { … }
        public String getTraceNotificationType() { … }
        public String getUsageNotificationType() { … }

        // JMX Usage MBean Object Name helper methods
        public static String getUsageMBeanProperties(String entityName) { … }
        public String getUsageMBeanProperties() { … }

        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
        public int compareTo(Object obj) { … }
}
```

- The `ALARM_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in alarm notifications generated by the Alarm MBean when a resource adaptor entity raises or clears an alarm. This notification type identifies the `NotificationSource` object in the `AlarmNotification` object as being of the type `ResourceAdaptorEntityNotification`.
- The `TRACE_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in trace notifications generated by the Trace MBean when a resource adaptor entity emits a trace message. This notification type identifies the `NotificationSource` object in the `TraceNotification` object as being of the type `ResourceAdaptorEntityNotification`.
- The `USAGE_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in usage notifications generated by a Usage MBean when a resource adaptor entity updates a usage parameter. This notification

type identifies the `NotificationSource` object in the `UsageNotification` object as being of the type `ResourceAdaptorEntityNotification`.

- The constructor.
  The constructor takes the following arguments:
  - The `entityName` argument specifies the name of a resource adaptor entity in the SLEE.
- The `getEntityName` method.
  This method returns the `entityName` argument of the constructor.
- The `getAlarmNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `ALARM_NOTIFICATION_TYPE` constant defined in this class.
- The `getTraceNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `TRACE_NOTIFICATION_TYPE` constant defined in this class.
- The `getUsageNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `USAGE_NOTIFICATION_TYPE` constant defined in this class.
- The `getUsageMBeanProperties` methods.
  These methods return a JMX Object Name property string that can be used as part of the complete Object Name of a Usage MBean for a resource adaptor entity. The zero-argument form of this method returns a property string for the `ResourceAdaptorEntityNotification` object that it is invoked on. The static one-argument form of this method returns a property string for the given `entityName` argument.
- The `equals` method.
  This method indicates whether the object specified by the `obj` argument is equal to this notification source. The specified object is equal to this notification source if the specified object is a `ResourceAdaptorEntityNotification` object encapsulating the same resource adaptor entity name as this.
- The `hashCode` method.
  This method provides a hash code value for the notification source.
- The `toString` method.
  This method provides a string representation for the notification source.
- The `compareTo` method.
  This method compares the object specified by the `obj` argument with this notification source for order.

14.16.5 `SubsystemNotification` class

Added in 1.1.

The `SubsystemNotification` class identifies an internal subsystem of the SLEE as the source of a notification. The SLEE specification does not define what the internal subsystems of a SLEE may be or how they should be named. It it up to the implementation of the SLEE to identify subsystems as appropriate. All SLEE internal subsystems that can cause notifications to be generated should be identified to management clients via the `getSubsystems` method on the `SleeManagementMBean` interface (refer Section 14.5.2)

The public interface of the `SubsystemNotification` class is as follows:

```
package javax.slee.management;
```

```
public final class SubsystemNotification implements NotificationSource {
        // constants
        public static final String ALARM_NOTIFICATION_TYPE =
                "javax.slee.management.alarm.subsystem";
        public static final String TRACE_NOTIFICATION_TYPE =
                "javax.slee.management.trace.subsystem";
        public static final String USAGE_NOTIFICATION_TYPE =
                "javax.slee.management.usage.subsystem";

        // additional JMX Object Name attribute of a Usage MBean for a vendor-defined
        // SLEE internal subsystem
        public static final String SUBSYSTEM_NAME_KEY = "subsystemName";

        // constructor
        public SubsystemNotification(String subsystemName) { … }

        // accessors
        public String getSubsystemName() { … }

        // JMX notification type accessors
        public String getAlarmNotificationType() { … }
        public String getTraceNotificationType() { … }
        public String getUsageNotificationType() { … }

        // JMX Usage MBean Object Name helper methods
        public static String getUsageMBeanProperties(String subsystemName) { … }
        public String getUsageMBeanProperties() { … }

        public boolean equals(Object obj) { … }
        public int hashCode() { … }
        public String toString() { … }
        public int compareTo(Object obj) { … }
}
```

- The `ALARM_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in alarm notifications generated by the Alarm MBean when an internal subsystem of the SLEE raises or clears an alarm. This notification type identifies the `NotificationSource` object in the `AlarmNotification` object as being of the type `SubsystemNotification`.
- The `TRACE_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in trace notifications generated by the Trace MBean when an internal subsystem of the SLEE emits a trace message. This notification type identifies the `NotificationSource` object in the `TraceNotification` object as being of the type `SubsystemNotification`.
- The `USAGE_NOTIFICATION_TYPE` constant.
  This constant defines the string that is used as the notification type in usage notifications generated by a Usage MBean when an internal subsystem of the SLEE updates a usage parameter. This notification type identifies the `NotificationSource` object in the `UsageNotification` object as being of the type `SubsystemNotification`.
- The constructor.
  The constructor takes the following arguments:
  - The `subsystemName` argument specifies the name of an internal subsystem in the SLEE.
- The `getSubsystemName` method.
  This method returns the `subsystemName` argument of the constructor.

- The `getAlarmNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `ALARM_NOTIFICATION_TYPE` constant defined in this class.
- The `getTraceNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `TRACE_NOTIFICATION_TYPE` constant defined in this class.
- The `getUsageNotificationType` method.
  This method implements the same method defined in the `NotificationSource` interface to return the value of the `USAGE_NOTIFICATION_TYPE` constant defined in this class.
- The `getUsageMBeanProperties` methods.
  These methods return a JMX Object Name property string that can be used as part of the complete Object Name of a Usage MBean for a SLEE internal subsystem. The zero-argument form of this method returns a property string for the `SubsystemNotification` object that it is invoked on. The static one-argument form of this method returns a property string for the given `subsystemName` argument.
- The `equals` method.
  This method indicates whether the object specified by the `obj` argument is equal to this notification source. The specified object is equal to this notification source if the specified object is a `SubsystemNotification` object encapsulating the same subsystem name as this.
- The `hashCode` method.
  This method provides a hash code value for the notification source.
- The `toString` method.
  This method provides a string representation for the notification source.
- The `compareTo` method.
  This method compares the object specified by the `obj` argument with this notification source for order.

14.17 SLEE Vendor extension data

Added in 1.1

The SLEE specification defines classes for a number of data types and notifications, instances of which are provided to management clients as the result of MBean method invocations or events occurring in the SLEE. In order to allow a SLEE vendor to supply additional information in these objects, above and beyond what is defined by the SLEE specification, many of these data types and notifications implement the `javax.slee.management.VendorExtensions` interface. This interface defines methods that allow arbitrary data to be included in the objects that implement this interface. This mechanism is provided to SLEE vendors to reduce the need to extend the SLEE API unnecessarily.

The following SLEE data-type classes implement the `VendorExtensions` interface:

- `javax.slee.management.Alarm`
- `javax.slee.management.Alarm.History`
- `javax.slee.management.ComponentDescriptor`
- `javax.slee.management.DeployableUnitDescriptor`
- `javax.slee.usage.SampleStatistics`

The following SLEE notification classes implement the `VendorExtensions` interface:

- `javax.slee.management.SleeStateChangeNotification`

- `javax.slee.management.ServiceStateChangeNotification`
- `javax.slee.management.ResourceAdaptorEntityStateChangeNotification`
- `javax.slee.management.AlarmNotification`
- `javax.slee.management.TraceNotification`
- `javax.slee.usage.UsageNotification`

A SLEE implementation that provides vendor-specific data to management clients should clearly document the type and nature of the data for each of the SLEE object types where it is included.

14.17.1 `VendorExtensions` interface

Added in 1.1

The public interface of the `VendorExtensions` interface is as follows:

```
package javax.slee.management;

public interface VendorExtensions {
        public void setVendorData(Object vendorData);
        public Object getVendorData();
}
```

- The `setVendorData` method.
  This method stores the object specified by the `vendorData` argument in the object implementing the `VendorExtensions` interface.
- The `getVendorData` method.
  This method returns the vendor-specific data object previously stored in the object implementing the `VendorExtensions` interface using the `setVendorData` method.

14.17.2 Controlling serialization and deserialization of vendor-specific data

Added in 1.1

Vendor-specific data may be included in SLEE objects for a number of reasons. For example, a SLEE vendor may wish to include additional information over and above what is defined by the SLEE specification to management clients receiving the data object or notification.

All classes that implement the `VendorExtensions` interface define the following static methods to control the serialization and deserialization of vendor-specific data within objects of their class (or subclass):

- The `enableVendorDataSerialization` method.
  This method enables the serialization of vendor-specific data for objects of the enclosing class (or subclass). This method is typically used by a SLEE implementation that wants to export vendor-specific data included in objects of this class to management clients.
  The serialization of vendor-specific data uses the standard Java serialization mechanism. This means that any vendor-specific data included in a SLEE object provided to a management client must be serializable by this standard mechanism if vendor data serialization has been enabled for the object's class.
- The `disableVendorDataSerialization` method.
  This method disables the serialization of vendor-specific data for objects of the enclosing class (or subclass).
- The `enableVendorDataDeserialization` method.
  This method enables the deserialization of vendor-specific data for objects of the enclosing class (or subclass). This method is typically used by a management client that wants to obtain any ven-

dor-specific data included in the serialization stream of objects of this class. If vendor-specific data deserialization is enabled, the client performing the deserialization must ensure that any necessary classes required to deserialize that vendor-specific data is available in the relevant classloader, otherwise deserialization may fail.

- The `disableVendorDataDeserialization` method.
  This method disables the deserialization of vendor-specific data for objects of the enclosing class (or subclass).

The default behavior of all classes implementing the `VendorExtensions` interface is for vendor-specific data serialization and deserialization to both be disabled. These features must explicitly be enabled via the appropriate method calls if their use is desired.

14.18 Security

Administrators acting in many different roles may manage the SLEE. For example, an Administrator may be a Service Administrator, a Profile Table Administrator, or a SLEE Administrator. Access to the management operations of the SLEE must be carefully controlled while taking into consideration the different roles of an Administrator. Different permissions can be granted to each role and the Administrator must be authenticated within a role before they can be authorized to invoke the methods associated with that role.

The SLEE specification does not define any security roles, security access protocol, or security mechanisms. It is the responsibility of the SLEE vendor to define a secure environment, either by defining a vendor specific mechanism, or by utilizing a standard mechanism such as Java Authentication and Authorization Service (JAAS). It is the responsibility of management client MLets to pass to the SLEE the relevant authentication principals. A management client should expect to receive `SecurityException`s from the SLEE if the Administrator does not have the appropriate permissions to execute a particular management operation.

Chapter 15 Resource Adaptors

Changed in 1.1: The 1.1 specification completes the 1.0 Resource Adaptor specification. The 1.0 specification introduced how Resource Adaptor Types[32] are specified and provided a DTD for Resource Adaptors. The 1.1 specification defines the contract and interfaces between the SLEE and Resource Adaptors such that Resource Adaptors may be developed by numerous parties. The intention for this contract is that Resource Adaptors can function in compliant 1.1 SLEE implementations.

Resources are entities that represent and interact with other systems outside the SLEE, such as network devices, protocol stacks, directories, and databases. These resources may or may not have existing Java APIs. Example resources with Java APIs include a SIP Stack supporting JAIN SIP (JSR32) for the Session Initiation Protocol (SIP) or databases supporting the JDBC API. These Java APIs define Java classes or interfaces to interact with the resource and to represent the events emitted by the resource.

Resource adaptors adapt particular resources to the requirements of the SLEE. A resource adaptor typically receives messages from an external system via a networking protocol and submits them as events to the SLEE.

The resource adaptor architecture defines a standardized packaging format and runtime environment for Resource Adaptors. The standardized packaging format allows a compliant SLEE to deploy a Resource Adaptor[33]. The standardized runtime environment allows the SLEE to manage the lifecycle of a Resource Adaptor, and provides mechanisms for Services and Resource Adaptors to interact. A Resource Adaptor using this environment may be deployed into any compliant SLEE, and execute correctly without customization.

15.1 Architecture overview

The SLEE Resource Adaptor architecture consists of APIs which are implemented by the SLEE, and APIs which are implemented by each Resource Adaptor. The following are key concepts behind the architecture:

- Resource adaptor type.
  A resource adaptor type defines an API and a behavioral contract that SBBs use to interact with a Resource. Resource adaptor types are independent of a particular Resource Adaptor implementation, allowing development of SBBs that can use multiple different implementations of the same resource adaptor type. Typically, a resource adaptor type is defined by an organization of collaborating SLEE or resource vendors, such as the SLEE expert group. An Administrator installs resource adaptor types in the SLEE.
- Resource Adaptor.
  A Resource Adaptor is an implementation of one or more resource adaptor types. There may be multiple Resource Adaptors available for a particular resource adaptor type, each providing the same contract to SBB developers. Typically, a Resource Adaptor is provided either by a resource vendor or a SLEE vendor to adapt a particular resource implementation to a SLEE. For example a vendor may provide a Resource Adaptor that adapts the SIP stack to the SLEE. An Administrator installs Resource Adaptors in the SLEE.
- Resource adaptor entity.
  A resource adaptor entity is the mapping, within the SLEE, of a particular resource as adapted by a Resource Adaptor. Multiple resource adaptor entities may be created from a single Resource Adaptor. For example a SIP Resource Adaptor may have multiple resource adaptor entities each responsible for a different instantiation of a SIP stack. Typically, an Administrator creates a resource adaptor entity from a Resource Adaptor installed in the SLEE and provides the appropriate configuration parameters. Configuration parameters enable a resource adaptor entity to "bind" to a particular resource.

[32] The contract between SBBs and Resources.
[33] Provided that the Resource Adaptor is packaged using the standardized packaging format.

- Resource adaptor object.
  Resource adaptor objects are the Java objects that the SLEE uses to interact with resource adaptor entities. A resource adaptor object is an instance of a class that implements the `ResourceAdaptor` interface defined by the Resource Adaptor architecture.
- SLEE Endpoint.
  The SLEE Endpoint is the interface used by a resource adaptor object to start Activities, fire Events on Activities, and end Activities.

Figure 21 depicts a UML meta-model of the Resource Adaptor architecture.

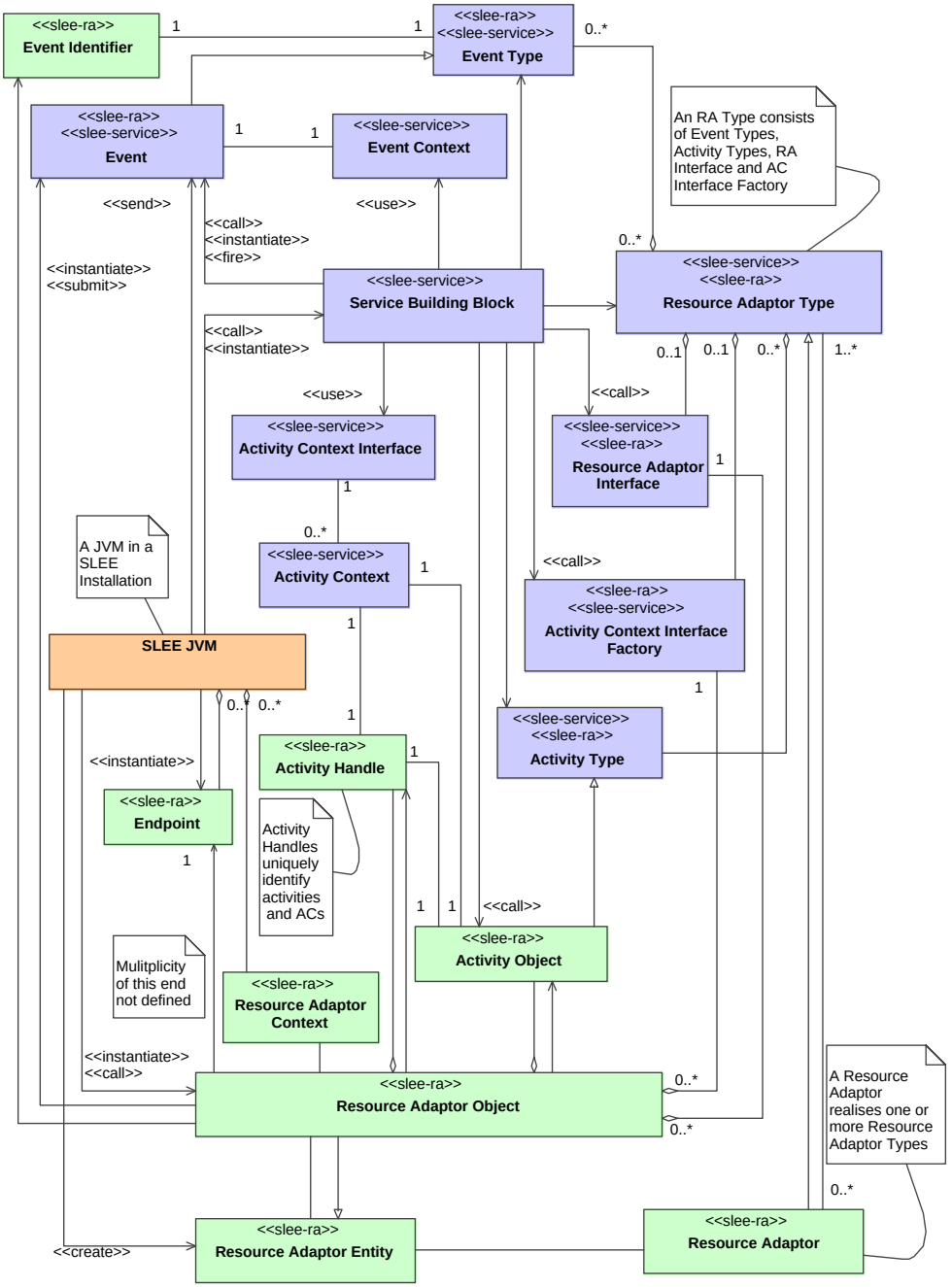

Figure 21 A meta-model of the Resource Adaptor architecture

15.2 Implementation Responsibility

The Resource Adaptor architecture defines various interfaces and contracts. The following table specifies whether the SLEE or each Resource Adaptor is responsible for implementing each interface or contract. An X in the "SLEE" column indicates that the SLEE is responsible for implementing the interface or contract whilst an X in the "Resource Adaptor" column indicates that each Resource Adaptor is responsible for implementing the interface or contract.

Interface or Contract	SLEE	Resource Adaptor
Resource Adaptor Type(s)		X
`javax.slee.resource.ResourceAdaptor`		X
`javax.slee.resource.ActivityHandle`		X
`javax.slee.resource.Marshaler`		X
Activity Context Interface Factories	X	
`javax.slee.resource.SleeEndpoint`	X	
`javax.slee.resource.ResourceAdaptorContext`	X	

15.3 Resource adaptor type

A resource adaptor type defines an API and a behavioral contract that SBBs use to interact with a resource[34]. Resource adaptor types are independent of a particular Resource Adaptor implementation, allowing the development of SBBs that can use multiple different implementations of the same resource adaptor type.

Each resource adaptor type must define the following:

- Event types.
  The resource adaptor type defines the event types that are fired by Resource Adaptors of this resource adaptor type (see Section 15.3.2).
- Activity types.
  For each identified Activity, the resource adaptor type must define a Java interface or class that represents the Activity (see Section 15.3.2). The resource adaptor type declares the interfaces and/or classes of these Activity objects. If a resource adaptor type maps an underlying resource's Activity that is represented by a primitive Java type to a Java object, then this mapping should be well-documented. Each Activity interface or class specified by a resource adaptor type must be defined in a named package, i.e. the interface or class must have a `package` declaration.
- Activity object end conditions.
  For each of the Activity interfaces and classes defined by the resource adaptor type, the conditions that end an Activity object of the resource adaptor type must be well documented. Typical conditions that end an Activity object are as follows:
    - The underlying Activity ends, e.g. a call terminates when the all of the participants hang up.

[34] SBBs invoke methods defined by a resource adaptor type.

  - A method that ends the Activity object is invoked, such as invoking the `endActivity` method on a `NullActivity` object (see Section 7.10).
  - A method is invoked that indirectly causes the underlying Activity to end (see Section 7.3.4).

- Activity Context Interface Factory interface.
  Each resource adaptor type that defines at least one Activity type defines an Activity Context Interface Factory interface (see Section 7.6.1). The Activity Context Interface Factory interface must be defined in a named package, i.e. the interface must have a `package` declaration. An object that implements the Activity Context Interface Factory interface can be bound into the JNDI component environment of an SBB using a `resource-adaptor-type-binding` element in the SBB's deployment descriptor. An SBB uses this object to access the Activity Context of a particular Resource Adaptor's Activity object[35]. As Resource Adaptors and the SLEE collaborate to implement this function, together they can validate the Activity object.
- A resource adaptor interface.
  Each resource adaptor type may optionally define a resource adaptor interface. This resource adaptor interface may be defined by the resource represented[36] or may be provided by the resource adaptor type[37]. The resource adaptor interface must be declared as `public` and must be defined in a named package, i.e. the interface must have a `package` declaration. An object that implements the resource adaptor interface is bound into the JNDI component environment of an SBB if it includes a `resource-adaptor-entity-binding` element in its deployment descriptor. This object allows the SBB to invoke the resource adaptor entity represented by the resource adaptor object. For example, the `JccProvider` interface is an appropriate resource adaptor interface for JCC.
  If a resource has more than one interface that should be made accessible via JNDI, then the resource adaptor type should define a wrapper interface that has accessor methods for these interfaces as the resource adaptor interface.
- Disallowed methods.
  The resource adaptor type may declare that a particular set of methods that are defined by the underlying resource API cannot be invoked by components running in the SLEE. For example, the resource adaptor type should disallow methods that affect event delivery and the location independence of SBBs. For example, the JCC resource adaptor type should specify that the `addConnectionListener`, `addCallListener`, and `setFilter` methods are disallowed. When a disallowed method is invoked, the resource adaptor entity must throw a `SecurityException`.

15.3.1 Resource adaptor type identifiers

Similar to SBB and Service, a resource adaptor type is uniquely identified by the name, vendor, and version of the resource adaptor type. Two resource adaptor types with same name, vendor, and version are considered to be the same resource adaptor type. Different resource adaptor types are assumed to be incompatible, meaning that one cannot be substituted for another.

15.3.2 Resource adaptor type deployment descriptor

A resource adaptor type deployment descriptor contains a `resource-adaptor-type` element. This element contains the following attributes and sub-elements:

- An `ignore-ra-type-event-type-check` attribute.
  This attribute determines if the SLEE will limit the event types accessible to the resource adaptor

[35] The resource adaptor entity that is responsible for this Activity object is an instance of a resource adaptor that belongs to this resource adaptor type.
[36] If the Resource provides a Java API.
[37] Regardless of whether or not the Resource provides a Java API.

to those referenced by the resource adaptor types implemented by the resource adaptor (see Section 15.10). The default value of this attribute if not specified is "False".

- A `description` element.
  This is an optional informational element.
- A `resource-adaptor-type-name` element, a `resource-adaptor-type-vendor` element, and a `resource-adaptor-type-version` element.
  These elements uniquely identify the resource adaptor type.
- Zero or more `library-ref` elements. *(Added in 1.1)*
  Each library required by the resource adaptor type must be identified by a `library-ref` element. Each `library-ref` element has the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - A `library-name` element, a `library-vendor` element, and a `library-version` element.
    These elements uniquely identify the library.
- A `resource-adaptor-type-classes` element.
  This element contains the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - Zero or more `activity-type` elements.
    The Java types of all the resource adaptor type's Activity objects must be declared. Each a`ctivity-type` element declares one of these Java types. Each a`ctivity-type` element contains the following sub-elements:
    - A `description` element.
      This is an optional informational element. This element should document the conditions that end Activity objects of this Java type.
    - An `activity-type-name` element.
      This element names the Java type of the Activity Objects. This must be a valid Java type name.
  - An `activity-context-interface-factory-interface` element.
    This element must be present if one or more activity types have been declared. It contains the following sub-elements:
    - A `description` element.
      This is an optional informational element.
    - An `activity-context-interface-factory-interface-name` element.
      This element names the Java interface of the Activity Context Interface Factory for the resource adaptor type.
  - A `resource-adaptor-interface` element.
    This element is optional. If present it names the Java type of the resource adaptor interface. Each Resource Adaptor of this resource adaptor type provides an implementation of this resource adaptor interface. An SBB can bind an object of this Java type into its JNDI component environment using a `resource-adaptor-entity-binding` element in the SBB's deployment descriptor. The `resource-adaptor-interface` element contains the following sub-elements:
    - A `description` element.
      This is an optional informational element.

        - A `resource-adaptor-interface-name` element.
          This element names the Java type of the resource adaptor interface.

- Zero or more `event-type-ref` elements.
  Each event type that the resource adaptor type may fire must have an `event-type-ref` element. An `event-type-ref` element references an event type. It contains following the sub-elements:
    - An `event-type-name` element, an `event-type-vendor` element, and an `event-type-version` element.
      These elements uniquely identify an event type declared in an `event-definition` element specified in another deployment descriptor. An `event-definition` element declares an event type (see Section 8.1.7)

15.3.3 Resource adaptor type jar file

The resource adaptor type jar file is the standard format for the packaging of one or more resource adaptor types. Resource adaptor types packaged in this format and included in a deployable unit may be deployed directly into a compliant SLEE. It must include the following:

- A resource adaptor type jar deployment descriptor.
    - The resource adaptor type jar deployment descriptor is stored with the name `META-INF/resource-adaptor-type-jar.xml` in the resource adaptor type jar file.
    - The root element of the resource adaptor type jar deployment descriptor is a `resource-adaptor-type-jar` element. This element contains the following sub-elements.
        - A `description` element.
          This is an optional informational element.
        - One or more `resource-adaptor-type` elements.
          Each of these elements is a resource adaptor type deployment descriptor as defined in Section 15.3.2.
- Class files of the resource adaptor types specified by the `resource-adaptor-type` elements of the `resource-adaptor-type-jar` element.
    - The resource adaptor type jar file must contain, either by inclusion or by reference, the class files of the resource adaptor types.
    - The resource adaptor type jar file must also contain, either by inclusion or by reference, the class files for all the classes and interfaces that the Java types of the resource adaptor types depend on, except J2SE classes and SLEE classes, classes contained in the event jar files of the referenced event types or classes contained in library jar files of the libraries referenced by the resource adaptor types. This includes their superclasses and superinterfaces, dependent classes, and the classes and interfaces used as method parameters, results, and exceptions.

15.3.4 Resource adaptor type jar file example

The following example illustrates a resource adaptor type jar file that declares a resource adaptor type.

The deployment descriptor for the example resource adaptor type jar file is as follows:

```
<resource-adaptor-type-jar>
        …
        <resource-adaptor-type>
                <description> … </description>
                <resource-adaptor-type-name>
                        JCC
                </resource-adaptor-type-name>
```

```
                <resource-adaptor-type-vendor>
                        javax.csapi.cc.jcc
                </resource-adaptor-type-vendor>
                <resource-adaptor-type-version>
                        1.1
                </resource-adaptor-type-version>
                <library-ref>
                        <description> … </description>
                        <library-name> JCC </library-name>
                        <library-vendor> javax.csapi.cc.jcc </library-vendor>
                        <library-version> 1.1 </library-version>
                </library-ref>
                <resource-adaptor-type-classes>
                        <description> … </description>
                        <activity-type>
                                <description> … </description>
                                <activity-type-name>
                                        javax.csapi.cc.jcc.JccCall
                                </activity-type-name>
                        </activity-type>
                        <activity-type>
                                <description> … </description>
                                <activity-type-name>
                                        javax.csapi.cc.jcc.JccConnection
                                </activity-type-name>
                        </activity-type>
                        <activity-context-interface-factory-interface>
                                <description> … </description>
                                <activity-context-interface-factory-interface-name>
                                javax.csapi.cc.jcc.slee.JccActivityContextInterfaceFactory
                                </activity-context-interface-factory-interface-name>
                        </activity-context-interface-factory-interface>
                        <resource-adaptor-interface>
                                <description> … </description>
                                <resource-adaptor-interface-name>
                                        javax.csapi.cc.jcc.JccProvider
                                </resource-adaptor-interface-name>
                        </resource-adaptor-interface>
                </resource-adaptor-type-classes>
                <event-type-ref>
                        <event-type-name>
                        javax.csapi.cc.jcc.JccConnectionEvent.CONNECTION_ALERTING
                        </event-type-name>
                        <event-type-vendor> javax.csapi.cc.jcc </event-type-vendor>
                        <event-type-version> 1.1 </event-type-version>
                </event-type-ref>
                …
        </resource-adaptor-type>
</resource-adaptor-type-jar>
```

The content of the resource adaptor type jar file is as follows:

```
META-INF/resource-adaptor-type-jar.xml
META-INF/MANIFEST.MF
…
javax/csapi/cc/jcc/slee/JccActivityContextInterfaceFactory.class
…
```

15.4 Resource adaptor

Changed in 1.1: The relationship between Resource Adaptor and resource adaptor type is defined such that a Resource Adaptor may now implement multiple resource adaptor types.

A Resource Adaptor is a particular implementation of one or more resource adaptor types. Each Resource Adaptor must identify the resource adaptor types it implements.

A Resource Adaptor connects a resource and the SLEE together. Resource Adaptors may start and end Activities, fire Events, and handle invocations from Services. The Resource Adaptor collaborates with the SLEE using the APIs and contracts defined by the Resource Adaptor architecture.

Resource Adaptors are packaged as a deployable unit. As such a deployment descriptor and packaging format are specified.

15.4.1 Resource adaptor configuration properties

Resource Adaptor configuration properties provide a standardized mechanism for configuration of a resource adaptor entity and its associated resource adaptor object(s). A Resource Adaptor specifies the configurable properties in its deployment descriptor. An Administrator may create a resource adaptor entity from a specific set of Resource Adaptor configuration property values by using a Resource Management MBean object (see section 14.12). A resource adaptor entity may also be reconfigured after it has been created by specifying a new set of configuration property values.

- A configuration property must be one of the following Java types: `String`, `Character`, `Integer`, `Boolean`, `Double`, `Byte`, `Short`, `Long`, and `Float`.
- If the Resource Adaptor Developer provides a value for a configuration property, the value must be a string that is valid for the constructor of the specified type that takes a single `String` parameter, or for `java.lang.Character`, a single character.
- A resource adaptor object accesses its configuration properties via the `javax.slee.resource.ConfigProperties` object provided to it in the `raConfigure` or `raConfigurationUpdate` methods (see Section 14.12.1).

15.4.2 Resource adaptor deployment descriptor

Each Resource Adaptor is declared by a Resource Adaptor deployment descriptor. Each Resource Adaptor deployment descriptor contains a `resource-adaptor` element. This element contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- A `resource-adaptor-name` element, a `resource-adaptor-vendor` element, and a `resource-adaptor-version` element.
  These elements uniquely identify the Resource Adaptor.
- One or more `resource-adaptor-type-ref` elements. *(Changed in 1.1: multiple resource adaptor types now supported)*
  Each resource adaptor type that is implemented by the resource adaptor is identified by a `resource-adaptor-type-ref` element Each `resource-adaptor-type-ref` element contains the following elements:
  - A `description` element.
    This is an optional informational element.
  - A `resource-adaptor-type-name` element, a `resource-adaptor-type-vendor` element, and a `resource-adaptor-type-version` element.
    These elements uniquely identify a resource adaptor type declared in a `resource-adaptor-type` element specified in another deployment descriptor. A `resource-adaptor-type` element declares a resource adaptor type (see Section 15.3.2).
- Zero or more `library-ref` elements. *(Added in 1.1)*
  Each library component required by the resource adaptor must be identified by a `library-ref` element. Each `library-ref` element has the following sub-elements:
  - A `description` element.
    This is an optional informational element.

  - A `library-name` element, a `library-vendor` element, and a `library-version` element.
    These elements uniquely identify the library component.

- A `resource-adaptor-class` element.
  This element contains sub-elements that identify the class that implements this Resource Adaptor.
  - A `supports-active-reconfiguration` attribute.
    This attribute determines whether or not resource adaptor objects of the resource adaptor class can be reconfigured by the Administrator when active. The value of this attribute is either "True" or "False". The default is "False", meaning resource adaptor objects may only be reconfigured when in the Inactive state.
  - A `description` element.
    This is an optional informational element.
  - A `resource-adaptor-class-name` element.
    This element identifies the fully qualified class name of the resource adaptor class (refer Section 15.6). This class implements the `javax.slee.resource.ResourceAdaptor` interface.
- Zero or more `config-property` elements. *(Added in 1.1)*
  Each configurable property of the Resource Adaptor must be identified by a `config-property` element. Each `config-property` element has the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - A `config-property-name` element.
    This element identifies the name of the configuration property used by the Resource Adaptor.
  - A `config-property-type` element.
    This element identifies the Java type of the configuration property used by the Resource Adaptor.
  - A `config-property-value` element.
    This element is optional and identifies the default value of the configuration property. The default value is used when creating a resource adaptor entity from the Resource Adaptor unless an alternative value is specified by the Administrator. If a default value is not specified in the deployment descriptor, then the Administrator must always supply a value for the configuration property when creating a resource adaptor entity from the Resource Adaptor.

15.4.3 Resource adaptor jar file

The resource adaptor jar file is the standard format for the packaging of one or more Resource Adaptors. It must include the following:

- A resource adaptor jar deployment descriptor.
  - The resource adaptor jar deployment descriptor is stored with the name `META-INF/resource-adaptor-jar.xml` in the resource adaptor jar file.
  - The root element of the resource adaptor jar deployment descriptor is a `resource-adaptor-jar` element. This element contains the following sub-elements.
    - A `description` element.
      This is an optional informational element.

- One or more `resource-adaptor` elements.
  Each of these elements is a Resource Adaptor deployment descriptor as defined in Section 15.4.2.
- A `security-permissions` element. *(Added in 1.1)*
  This element is optional and contains sub-elements that identify security permissions of the resource adaptor classes. This element contains the following sub-elements.
  - A `description` element.
    This is an optional informational element.
  - A `security-permission-spec` element.
    This element identifies the security permission policies used by the resource adaptor classes. The syntax of this element is defined at the following URL:
    `http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html#FileSyntax`.
    Note: If the `codeBase` argument is not specified for a `grant` entry in the provided security permission specification, the SLEE assumes the code base to be the Resource Adaptor component jar file, and the security permissions specified in the `grant` entry are granted to all classes loaded from the Resource Adaptor component jar file, i.e. they apply to all Resource Adaptor components defined in the Resource Adaptor component jar file. The security permissions are not granted to classes loaded from any other dependent component jar required by the Resource Adaptors defined in the Resource Adaptor deployment descriptor. If the `codeBase` argument is specified for a `grant` entry, the specified path is taken to be relative to the root directory of the Resource Adaptor component jar within the deployable unit jar, but its use is otherwise undefined by the SLEE specification.

- Class files of the Resource Adaptors specified by the `resource-adaptor` elements of the `resource-adaptor-jar` element.
  - The resource adaptor jar file must contain, either by inclusion or by reference, the class files of the Resource Adaptors.
  - The resource adaptor jar file must also contain, either by inclusion or by reference, the class files for all the classes and interfaces that the Java types that the Resource Adaptors depend on, except J2SE classes, SLEE classes, classes contained in the resource adaptor type jar files of the resource adaptor types of the Resource Adaptors, classes contained in library jar files of the libraries referenced by the Resource Adaptor, classes contained in the Profile Specification jar files of the Profile Specifications referenced by the Resource Adaptor, or classes contained in the event jar files of the event types referenced by the resource adaptor types of the Resource Adaptors. This includes their superclasses and superinterfaces, dependent classes, and the classes and interfaces used as method parameters, results, and exceptions.

15.4.4 Resource adaptor jar file example

The following example illustrates a resource adaptor jar file that declares a resource adaptor.

The deployment descriptor for the example resource adaptor jar file is as follows:

```
<resource-adaptor-jar>
       <description> … </description>
       …
       <resource-adaptor>
              <description> … </description>
```

```
                <resource-adaptor-name> Foo JCC </resource-adaptor-name>
                <resource-adaptor-vendor> com.foo </resource-adaptor-vendor>
                <resource-adaptor-version> 2.6 </resource-adaptor-version>
                <resource-adaptor-type-ref>
                        <description> … </description>
                        <resource-adaptor-type-name>
                                JCC
                        </resource-adaptor-type-name>
                        <resource-adaptor-type-vendor>
                                javax.csapi.cc.jcc
                        </resource-adaptor-type-vendor>
                        <resource-adaptor-type-version>
                                1.1
                        </resource-adaptor-type-version>
                </resource-adaptor-type-ref>
                <resource-adaptor-classes>
                        <resource-adaptor-class supports-active-reconfiguration="True">
                                <description> … </description>
                                <resource-adaptor-class-name>
                                        com.foo.FooResourceAdaptor
                                </resource-adaptor-class-name>
                        </resource-adaptor-class>
                        …
                </resource-adaptor-classes>
                <config-property>
                        <description> … </description>
                        <config-property-name>
                                host
                        </config-property-name>
                        <config-property-type>
                                java.lang.String
                        </config-property-type>
                        <config-property-value>
                                localhost
                        </config-property-value>
                </config-property>
                …
        </resource-adaptor>

        <security-permissions>
                <description> … </description>
                <security-permission-spec>
                        grant {
                                permission java.lang.RuntimePermission "modifyThreadGroup";
                        };
                </security-permission-spec>
        </security-permissions>
        …
</resource-adaptor-jar>
```

The content of the resource adaptor jar file is as follows:

```
META-INF/resource-adaptor-jar.xml
META-INF/MANIFEST.MF
…
com/foo/FooResourceAdaptor.class
…
```

15.5 Resource adaptor entity

A resource adaptor entity is logical entity that represents the persistent state of a configured Resource Adaptor. Resource Adaptor entities have a state machine (see section 15.5.1). Multiple resource adaptor entities may be instantiated from a single Resource Adaptor. Typically, an administrator creates a resource adaptor entity from a Resource Adaptor installed in the SLEE and manages that entity using the `ResourceManagementMBean` interface (see section 14.12). Each resource adaptor entity is created with a set of properties that allow that entity to bind to a particular resource. These properties are part of the state of the resource adaptor entity. The properties that are configured for a resource adaptor entity are specific to the

resource adaptor implementation. For example, if a resource adaptor entity uses TCP/IP and accepts incoming connections one of the configurable parameters of the resource adaptor entity may be port number.

15.5.1 Resource adaptor entity life cycle

The resource adaptor entity state machine is modeled similarly to the state machine for a Service. The resource adaptor entity state machine and SLEE state machine together drive the resource adaptor object state transitions (see section 15.7.1). For example if a set of management operations are performed on a resource adaptor entity via the `ResourceManagementMBean` there may be sequences of lifecycle operations invoked on the resource adaptor object(s) instantiated by the SLEE for that resource adaptor entity.

A resource adaptor entity can be in one of the following three operational states (see Figure 22).

- Inactive.
  A resource adaptor entity enters the Inactive state when it has been successfully created in the SLEE. An Inactive resource adaptor entity is ready to be activated.
- Active.
  The resource adaptor entity has been activated. If the SLEE is in the Running state, resource adaptor objects associated with the resource adaptor entity can create new activities, submit events on activities, and end activities. If the SLEE is in the Stopping state, resource adaptor objects associated with the resource adaptor entity cannot start new activities, but may submit events on existing activities and end activities.
- Stopping.
  The resource adaptor entity is being deactivated. However, some activities created by the resource adaptor objects associated with the resource adaptor entity may still exist in the SLEE and have not completed their processing. The SLEE is waiting for these activities to end. Once all activities owned by resource adaptor objects of the resource adaptor entity have ended, the SLEE will transition the resource adaptor entity back to the Inactive state.

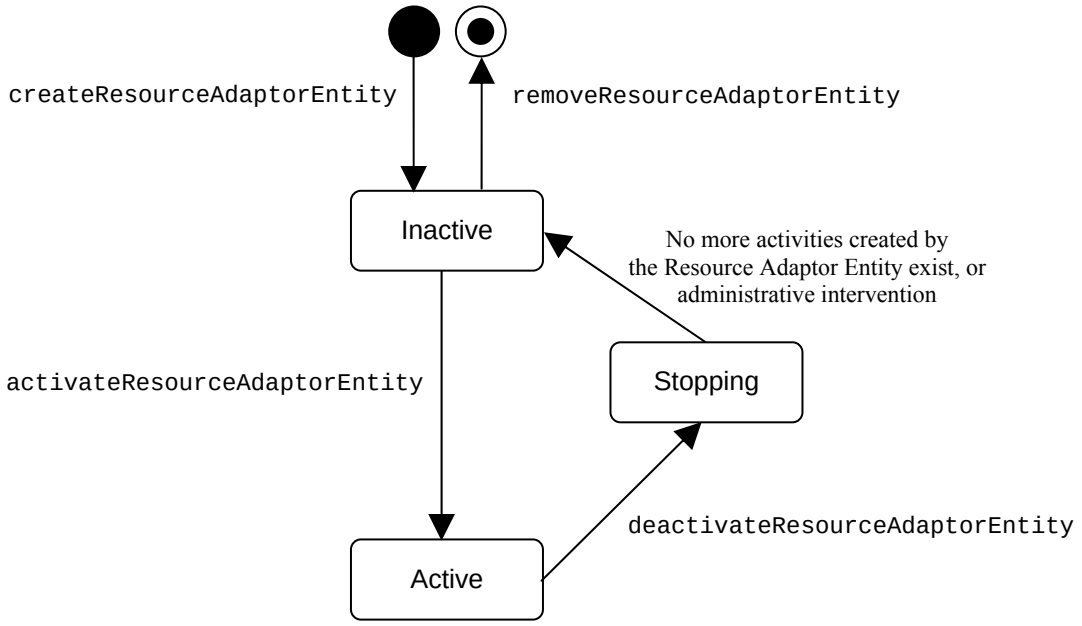

Figure 22 Resource Adaptor Entity Lifecycle
All specified methods are on the `ResourceManagementMBean` interface

The operational state of resource adaptor entities is persistent, i.e. the SLEE stores the state of the resource adaptor entities. If the SLEE is shut down and then restarted, the SLEE will restore the resource adaptor entities to their previous operational state. It is expected that non-volatile storage is used to store the operational state of the resource adaptor entities.

15.6 Resource adaptor class

The resource adaptor class has the following requirements:

- The resource adaptor class must be defined in a named package, i.e. the class must have a `package` declaration.
- The class must implement, either directly or indirectly, the `javax.slee.resource.ResourceAdaptor` interface (see Section 15.12).
- The class must be defined as `public` and cannot be `abstract` or `final`.
- The class must have a public no-argument constructor.

15.7 Resource adaptor object

A resource adaptor object is an instance of a resource adaptor class. It has a lifecycle. Resource adaptor objects are the Java objects that execute on behalf of resource adaptor entities and are provided with the persistent configuration information of a resource adaptor entity. The class of a Resource adaptor object must implement the `ResourceAdaptor` interface defined by the SLEE.

A resource adaptor object may be considered to be either configured or unconfigured. A configured resource adaptor object is in the Inactive, Active, or Stopping state. An unconfigured resource adaptor object is in the Unconfigured state. There is at most one configured resource adaptor object present in a Java Virtual Machine for each resource adaptor entity. The SLEE may create additional unconfigured resource adaptor objects for a resource adaptor entity if required for the purposes of invoking the `raVerifyConfiguration` method only (see Section 15.12.2.1).

15.7.1 Resource adaptor object life cycle

A resource adaptor object can be in one of the following four operational states (see Figure 5).

- Unconfigured state.
  The resource adaptor object has been created and provided with a `ResourceAdaptorContext` object, but has no configuration information for the resource adaptor entity that it was created for.
- Inactive state.
  The resource adaptor object has been configured for the resource adaptor entity. It is ready to work on behalf of the resource adaptor entity but is not yet creating Activities or firing Events.
- Active state.
  The resource adaptor object has been activated, i.e. it is running. The resource adaptor object can create new Activities, submit Events on Activities, and end Activities on behalf of the resource adaptor entity.
- Stopping state.
  The resource adaptor object is being deactivated. The resource adaptor object continues to manage any remaining Activities that are owned by the resource adaptor object. It can submit Events on Activities and end Activities, but cannot start new Activities.

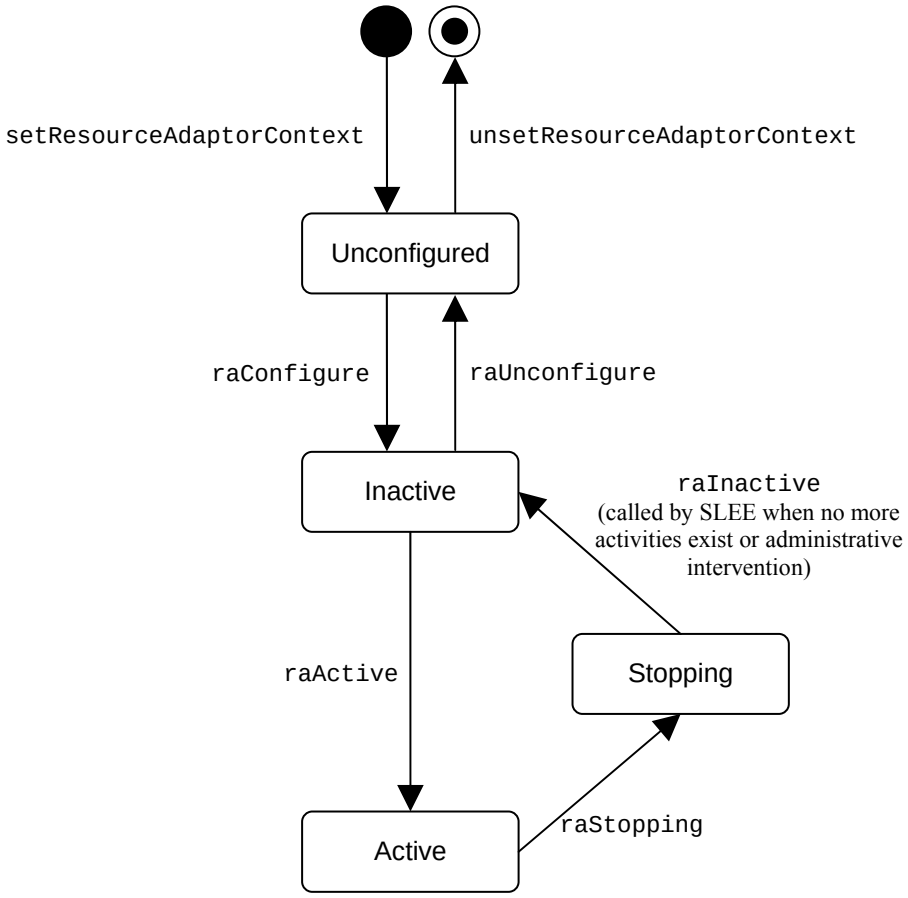

Figure 23 Resource Adaptor Object Lifecycle
All specified methods are on the `ResourceAdaptor` interface

The following steps describe the life cycle of a resource adaptor object.

- A resource adaptor object initially enters the Unconfigured state after it is constructed and returns from the `setResourceAdaptorContext` method. A resource adaptor object in the Unconfigured state is viewed by the SLEE as having been constructed and provided with an environment context, but is not yet provided with any configuration information.
    - In the Unconfigured state, a resource adaptor object may not start Activities or submit Events. The resource adaptor object may obtain references from its `ResourceAdaptorContext` object but should not allocate threads or other system resources in this state.
- A resource adaptor object transitions from the Unconfigured state to the Inactive state when it returns from its `raConfigure` method. A resource adaptor object in the Inactive state is viewed by the SLEE as having been initialized and configured for the resource adaptor entity, and is idle (i.e. it has no active Activities and is not firing Events to the SLEE).
    - In the Inactive state a resource adaptor object may not start Activities or submit Events. It is recommended that the resource adaptor object does not allocate threads or other system resources unnecessary in this state.

- A resource adaptor object transitions from the Inactive state to the Active state when both the following conditions become true:
    1. The SLEE is in the Running state.
    2. The resource adaptor entity of the resource adaptor object is in the Active state.

  The SLEE invokes the `raActive` method on a resource adaptor object to transition the resource adaptor object to the Active state. A resource adaptor object in the Active state is viewed by the SLEE as an actively executing object which may allocate and deallocate systems resources as required.
    - In the Active state a resource adaptor object may start, submit Events on, and end Activities.

  The resource adaptor object can consider itself in the Active state, and hence have the ability to start, submit Events on, and end Activities, from the moment it receives the `raActive` method invocation.
- A resource adaptor object transitions from the Active state to the Stopping state when one of the following conditions becomes true:
    1. The SLEE transitions to the Stopping state.
    2. The resource adaptor entity of the resource adaptor object transitions to the Stopping state.

  The SLEE invokes the `raStopping` method on a resource adaptor object to transition the resource adaptor object to the Stopping state. A resource adaptor object in the Stopping state is viewed by the SLEE as an Active object that is shutting down.
    - In the Stopping state a resource adaptor object may submit Events on existing Activities and end Activities, however it may not start new Activities.

  The resource adaptor object must consider itself in the Stopping state, and hence be disallowed from starting new Activities, once the `raStopping` method returns. The resource adaptor object should modify any internal state required so that the resource adaptor object does not try to create new Activities once it is in the Stopping state.
- A resource adaptor object transitions from the Stopping to the Inactive state after all Activities being managed by the resource adaptor object have ended. The SLEE invokes the `raInactive` method on a resource adaptor object to transition it from the Stopping state to the Inactive state.
    - The resource adaptor object should deallocate any threads and other system resources it has previously allocated as it transitions to the Inactive state.
- The SLEE invokes the `raUnconfigure` method on a resource adaptor object to transition it from the Inactive state to the Unconfigured state. For example, this method would be invoked on existing resource adaptor objects of a resource adaptor entity when the Administrator chooses to remove the resource adaptor entity from the SLEE. The resource adaptor object should release any system resources still allocated during the `raUnconfigure` method invocation.
- When the SLEE no longer requires a resource adaptor object that is in the Unconfigured state, it invokes the `unsetResourceAdaptorContext` method to notify the resource adaptor object that it will become a candidate for garbage collection. The resource adaptor object must perform any final cleanup operations necessary to release system resources still held.

15.8 Activities and Activity Handles

A Resource Adaptor uses an Activity Handle to identify each Activity that it starts, ends, or fires Events on to the SLEE. An Activity Handle is a non-transactional distributable reference to an Activity object that defines a specific contract for equality. Activity Handles are used to identify Activities to the SLEE for the following reasons:

- The Java class of an arbitrary Activity object may not have a specific contract for equality, thereby making them incomparable in the SLEE.
- The Java class of an arbitrary Activity object may not have a specific contract for serialization, thereby making their distribution within a cluster of SLEE nodes difficult or impossible.
- The Activity Context Interfaces used by SBBs to identify Activities are transactional. A Resource Adaptor may be interfacing with non-transactional resources therefore a non-transactional identifier needs to be used to identify Activities in the Resource Adaptor.

Each Activity Handle uniquely identifies an Activity. There is a 1:1 relationship between an Activity Handle and an Activity.

Within the SLEE, an Event is always delivered on an Activity Context. There is a 1:1 relationship between an Activity Handle (belonging to a particular resource adaptor entity) and an Activity Context. The Activity Handle that a Resource Adaptor uses to identify the Activity an Event is fired on is used by the SLEE to identify the Activity Context that should be used to deliver the Event to SBBs with.

It is not necessary for all Activity Handles in the SLEE at any one time to be unique. More specifically, two resource adaptor entities (of the same or different Resource Adaptor) may generate Activity Handles at the same time that are considered equal. Since the SLEE is aware of which resource adaptor entity a particular Activity Handle belongs to, the SLEE is able to differentiate between two otherwise equal Activity Handles. A single resource adaptor entity, however, must always generate an unique Activity Handle for each unique Activity.

An Activity Handle implemented by a Resource Adaptor must implement the `javax.slee.resource.ActivityHandle` interface.

15.8.1 `ActivityHandle` interface

The `ActivityHandle` interface is defined as follows:

```
package javax.slee.resource;

public interface ActivityHandle {
        // methods
        public boolean equals(Object obj);
        public int hashCode();
}
```

Implementations of the `ActivityHandle` interface are expected to implement the `equals` and `hashcode` methods:

- The `equals` method.
  The implementation class of an Activity Handle object must override the `Object.equals` method. The new `equals` method must ensure that the implementation of the method is reflexive, symmetric, transitive and for any non-null value `X`, `X.equals(null)` returns `false`. See `java.lang.Object` for more information.
- The `hashCode` method.
  The implementation class of an Activity Handle object must override the `Object.hashCode` method. The new `hashCode` method does not need to provide hash codes that are unique but does need to provide hash codes that are well distributed. A well-distributed hash code algorithm reduces but does not eliminate the number of times different Activity Handle objects evaluate to the same hash code. When different Activity Handle objects evaluate to the same hash code they may be stored together and differentiated by their `equals` method.

15.9 Event types and the `FireableEventType` interface

The SLEE uses event type component identifiers to identify the event types that are installed in the SLEE. However Resource Adaptors do not use the same component identifiers to identify the event types of Events that they fire or receive event processing callbacks for. Rather, a Resource Adaptor uses a

`javax.slee.resource.FireableEventType` object to indicate the event type of an Event. `FireableEventType` is an SLEE-defined interface implemented by the SLEE to contain any data necessary to efficiently process fired events.

A Resource Adaptor uses the Event Lookup Facility (see Section 13.7) to obtain the `FireableEventType` object for an event type specified by an event type component identifier. A Resource Adaptor may only request `FireableEventType` objects from the Event Lookup Facility for event types that the Resource Adaptor may fire, as determined by the resource adaptor types implemented by the Resource Adaptor, unless event type checking has been disabled for the Resource Adaptor (see Section 15.10).

`FireableEventType` objects are used in the following interfaces:

- The `SleeEndpoint` interface (see Section 15.14)
  They are used to identify the event type of an Event that is fired to the SLEE using the `SleeEndpoint` interface.
- The `ResourceAdaptor` interface (see Section 15.12)
  They are used to identify the event type of an Event in the event-related callback methods initiated by the SLEE on the `ResourceAdaptor` interface.
- The `Marshaler` interface (see Section 15.15)
  They are used to identify the event type of an Event that the SLEE is requesting to be marshaled or unmarshaled using the `Marshaler` interface.

15.10 Relaxing resource adaptor type event type checking

By default the SLEE limits the visibility of event type information provided by the SLEE to a resource adaptor object to the event types referenced by the resource adaptor types implemented by the resource adaptor. This policy helps to expose potential bugs in Resource Adaptor code and also eliminates the need for a Resource Adaptor to handle information or receive callbacks for event types it does not understand or care for.

The following sections of the Resource Adaptor architecture include an event type checking policy:

- The Event Lookup Facility (see Section 13.7) will check that a Resource Adaptor only attempts to lookup event type information for event types referenced by the resource adaptor types implemented by the Resource Adaptor.
- A SLEE Endpoint will only allow a Resource Adaptor to fire events of event types referenced by the resource adaptor types implemented by the Resource Adaptor.
- The Service Lookup Facility (see Section 13.8) will only return Service event type information for the event types referenced by the resource adaptor types implemented by the Resource Adaptor.
- The SLEE will check the event types that can be received by a Service and will only make event filter method callbacks (see Section 15.12.4) to a Resource Adaptor for that Service if the Service has an event handler for at least one event type referenced by the resource adaptor types implemented by the Resource Adaptor.

In some circumstances it may be required for a Resource Adaptor to have the capability to fire events of arbitrary types, or to have access to all event type information available in the SLEE. The `ignore-ra-type-event-type-check` attribute of the `resource-adaptor` element in the Resource Adaptor deployment descriptor is used to control whether the SLEE performs event type checking for resource adaptor entities of a Resource Adaptor.

If the value of this attribute is “False” then the SLEE performs the described event type checks for resource adaptor entities of the Resource Adaptor.

If the value of this attribute is “True” then the SLEE disables all event type checks for resource adaptor entities of the Resource Adaptor. These resource adaptor entities have full access to all event type information in the SLEE, and may fire events of any event type installed in the SLEE.

The default value of the `ignore-ra-type-event-type-check` attribute, if it is not specified in the deployment descriptor, is "False".

15.11 Transactions

Resource adaptor objects have access to a SLEE transaction manager implementation via the `ResourceAdaptorContext` interface. For more information on the topic of transactions and Resource Adaptors see section 9.14 in the Transactions chapter. In general a Resource Adaptor should commit or rollback all transactions that it begins, and only commit or rollback transactions that it begins. The SLEE is responsible for completing any transaction that the SLEE begins.

15.11.1 Transaction Context Propagation

When an SBB invokes any method declared by a resource adaptor type (which, for example, could be a method on an Event object, or Activity object or a resource adaptor interface object bound into the SBB's JNDI environment) the transaction context which the SBB is executing within is propagated to the resource adaptor object. This means that the resource adaptor object is able to use the `SleeTransactionManager` interface to get a reference to a `SleeTransaction` object representing the current transaction context.

When a Resource Adaptor invokes any method on the `SleeEndpoint` the transaction context within which the Resource Adaptor is executing is propagated to the implementation of the `SleeEndpoint`. The `SleeEndpoint` interface defines which methods require a transaction context.

When a Resource Adaptor invokes any method on the Profile subsystem the transaction context within which the Resource Adaptor is executing is propagated to the Profile subsystem.

15.12 `ResourceAdaptor` interface

The `ResourceAdaptor` interface must be implemented by all resource adaptors. This interface defines the necessary functionality required by the SLEE to manage resource adaptor objects.

The `ResourceAdaptor` interface contains methods related to the following areas:

- Lifecycle methods.
  These methods allow the SLEE to manage the lifecycle of the resource adaptor object.
- Configuration management methods.
  These methods allow the SLEE to manage configuration changes to the resource adaptor object.
- Inteface access methods.
  These methods provide the SLEE with access to objects that implement additional interfaces provided by the Resource Adaptor. These methods allow the SLEE to properly configure both the application and SLEE environment necessary to support the resource adaptor entity.
- Event filter methods.
  These methods inform a resource adaptor object which event types may be processed by Services installed in the SLEE. These methods allow a resource adaptor object to optimize which events it fires to the SLEE.
- Mandatory callback methods.
  These methods are invoked by the SLEE. The resource adaptor object is expected to fulfill the contract for these methods.
- Optional callback methods.
  These methods are only invoked by the SLEE if the resource adaptor object requests that they are invoked. The resource adaptor object requests invocation of these methods via the use of the ActivityFlags when starting an Activity or the EventFlags when firing an event.

The `ResourceAdaptor` interface is as follows:

```
package javax.slee.resource;
```

```
import javax.slee.Address;
import javax.slee.ServiceID;

public interface ResourceAdaptor {
        // lifecycle methods
        public void setResourceAdaptorContext(ResourceAdaptorContext context);
        public void unsetResourceAdaptorContext();
        public void raConfigure(ConfigProperties properties);
        public void raUnconfigure();
        public void raActive();
        public void raStopping();
        public void raInactive();

        // configuration management methods
        public void raVerifyConfiguration(ConfigProperties properties)
                throws InvalidConfigurationException;
        public void raConfigurationUpdate(ConfigProperties properties);

        // interface access methods
        public Object getResourceAdaptorInterface(String className);
        public Marshaler getMarshaler();

        // event filtering methods
        public void serviceActive(ReceivableService serviceInfo);
        public void serviceStopping(ReceivableService serviceInfo);
        public void serviceInactive(ReceivableService serviceInfo);

        // mandatory callback methods
        public void queryLiveness(ActivityHandle handle);
        public Object getActivity(ActivityHandle handle);
        public ActivityHandle getActivityHandle(Object activity);
        public void administrativeRemove(ActivityHandle handle);

        // optional callback methods
        public void eventProcessingSuccessful(ActivityHandle handle,
                                FireableEventType eventType, Object event, Address address,
                                ReceivableService service, int flags);
        public void eventProcessingFailed(ActivityHandle handle,
                                FireableEventType eventType, Object event, Address address,
                                ReceivableService service, int flags,
                                FailureReason reason);
        public void eventUnreferenced(ActivityHandle handle, FireableEventType eventType,
                                Object event, Address address, ReceivableService service,
                                int flags);
        public void activityEnded(ActivityHandle handle);
        public void activitySleeGC(ActivityHandle handle);
}
```

Unless otherwise specified, all methods on the `ResourceAdaptor` interface are executed in an unspecified transaction context. In other words, the SLEE specification does not define if they are invoked with or without a transaction context. If a method requires use of transactions then it may use the `SleeTransactionManager` interface to determine if a transaction is active, and start a transaction if necessary. A method must return with the same transaction context as it was invoked with (if any). If the SLEE detects that a different transaction context is associated with the thread after the method returns, the SLEE will log this condition and rollback the transaction.

The SLEE will invoke lifecycle and configuration management methods on a single resource adaptor object in serial, i.e. the resource adaptor object does not have to be written to handle concurrent invocations of these methods. All other methods may be invoked by the SLEE at any time while the resource adaptor object is in the Inactive, Active, or Stopping state, therefore these methods should be written to handle concurrent invocations.

15.12.1 Resource adaptor object lifecycle methods

The lifecycle methods are invoked on a resource adaptor object to inform it that the resource adaptor object has changed lifecycle state.

A resource adaptor object must assume that any lifecycle method invoked on it is done so in a thread owned by the SLEE. The resource adaptor object must return in a timely manner from any lifecycle methods invoked on it[38]. If a resource adaptor needs to perform long-running operations or similar such work as part of a lifecycle operation, the resource adaptor object should use its own worker thread to do this.

The lifecycle methods throw no checked exceptions. However, if a lifecycle method returns by throwing an unchecked exception, the SLEE must log the exception. Regardless of whether or not a lifecycle method returns by throwing an unchecked exception, the resource adaptor object invoked still transitions from one lifecycle state to another as appropriate for the method invoked. In other words, lifecycle state transitions of a resource adaptor object occur regardless of the outcome of the lifecycle method invoked on the resource adaptor object. A resource adaptor object must be careful about throwing unchecked exceptions from its lifecycle methods that may leave it in an unpredictable state.

15.12.1.1 `ResourceAdaptor` interface `setResourceAdaptorContext` method

This method is invoked by the SLEE after a new instance of a resource adaptor object is created for a resource adaptor entity. It uses this method to pass a `ResourceAdaptorContext` object to the resource adaptor object. If the resource adaptor object needs to use the `ResourceAdaptorContext` object during its lifetime, it should keep a reference to the `ResourceAdaptorContext` object in an instance variable.

This method will be invoked on a resource adaptor object before any other method defined by the `ResourceAdaptor` interface is invoked on the same resource adaptor object by the SLEE.

- The `setResourceAdaptorContext` method must be declared as `public` and cannot be `static` or `abstract`.

This method takes the following argument:

- The `context` argument references a `ResourceAdaptorContext` object (see Section 15.13). The resource adaptor object can obtain references to a `SleeEndpoint` object and various SLEE Facilities from the `ResourceAdaptorContext` object.

15.12.1.2 `ResourceAdaptor` interface `unsetResourceAdaptorContext` method

This method is invoked by the SLEE before terminating the life of the resource adaptor object. The resource adaptor object should use this method to free any remaining state or release any remaining resources that it is using. After this method returns, the resource adaptor object becomes a candidate for garbage collection by the SLEE.

- The `unsetResourceAdaptorContext` method must be declared as `public` and cannot be `static` or `abstract`.

15.12.1.3 `ResourceAdaptor` interface `raConfigure` method

This method is invoked by the SLEE when it needs to configure a resource adaptor object. This method may be invoked when a resource adaptor entity is created by the Administrator, during a SLEE reboot, or at any other time when the SLEE needs to create and configure a new resource adaptor object for a resource adaptor entity.

[38] The Resource adaptor is expected to return from these methods as soon as possible.

In the case where a resource adaptor entity is being created by the Administrator, before this method is invoked, the `raVerifyConfiguration` method (see Section 15.12.2.1) is invoked on a resource adaptor object of the same resource adaptor entity[39] to validate the configuration properties supplied by the Administrator. If the `raVerifyConfiguration` method returns without throwing an exception, the SLEE assumes that the configuration properties are valid for the Resource Adaptor and proceeds to invoke the `raConfigure` method with the same set of configuration properties in order to configure the resource adaptor object.

During this method, the resource adaptor object is responsible for inspecting the configuration properties it is provided with to determine any appropriate internal action necessary to realize the requested configuration.

- The `raConfigure` method must be declared as `public` and cannot be `static` or `abstract`.
- It is recommended that the resource adaptor object not allocate system resources during this method that it does not need while in the Inactive state. For example, starting threads or opening sockets to remote resources are generally unnecessary when the resource adaptor object is in the Inactive state.

This method takes the following argument:

- The `properties` argument is a `javax.slee.resource.ConfigProperties` object that contains the set of configuration properties for the resource adaptor entity associated with the resource adaptor object. A resource adaptor object may store a reference to this object in an instance variable if desired. For more information on `ConfigProperties` object refer to section 14.12.1.

After this method returns, the SLEE must not modify the `ConfigProperties` object passed to the resource adaptor object as the argument to this method, as ownership of the `ConfigProperties` object is transferred to the resource adaptor object as a result of this method invocation.

15.12.1.4 `ResourceAdaptor` interface `raUnconfigure` method

This method is invoked by the SLEE when a resource adaptor object is no longer required, for example because its associated resource adaptor entity is being removed by the Administrator. During this method the resource adaptor object is expected to release any system resources it has allocated in the corresponding `raConfigure` method.

- The `raUnconfigure` method must be declared as `public` and cannot be `static` or `abstract`.

15.12.1.5 `ResourceAdaptor` interface `raActive` method

This method is invoked by the SLEE to notify a resource adaptor object that it is transitioning to the Active state. Before this method is invoked, the SLEE modifies any internal state necessary so that the `SleeEndpoint` object associated with the resource adaptor object can be used to start new Activities, fire Events on those Activities, and end Activities. The resource adaptor object may start Activities and fire Events on those Activities during the `raActive` method if required. During this method the resource adaptor object may allocate any system resources required to allow it to interact with the underlying resource.

- The `raActive` method must be declared as `public` and cannot be `static` or `abstract`.

15.12.1.6 `ResourceAdaptor` interface `raStopping` method

The method is invoked by the SLEE to notify a resource adaptor object that it is transitioning to the Stopping state. During this method the resource adaptor object should update any internal state required such

[39] At the discretion of the SLEE, this may or may not be the same Resource Adaptor object receiving the `raConfigure` callback method.

that it does not attempt to start any new Activities once this method returns. Once this method does return, the SLEE modifies any of its own internal state necessary so that the `SleeEndpoint` object associated with the resource adaptor object no longer allows new activities to be started.

- The `raStopping` method must be declared as `public` and cannot be `static` or `abstract`.

15.12.1.7 `ResourceAdaptor` interface `raInactive` method

This method is invoked by the SLEE to notify the resource adaptor object that the SLEE is not aware of any remaining activities owned by the resource adaptor entity of the resource adaptor object and thus is transitioning the resource adaptor object from the Stopping state to the Inactive state. Once this method returns, the SLEE modifies any internal state necessary to fully disable the `SleeEndpoint` object associated with the resource adaptor object of any function.

- The `raInactive` method must be declared as `public` and cannot be `static` or `abstract`.

15.12.2 Configuration management methods

The configuration management methods are used to manage configuration changes to a resource adaptor entity.

A resource adaptor object must assume that any configuration management method invoked on it is done so in a thread owned by the SLEE. The resource adaptor object must return in a timely manner from any configuration management methods invoked on it.

15.12.2.1 `ResourceAdaptor` interface `raVerifyConfiguration` method

This method is invoked by the SLEE in order to provide a Resource Adaptor with the opportunity to verify that the configuration properties specified by the Administrator for a resource adaptor entity of the Resource Adaptor are valid. During this method the invoked resource adaptor object should inspect the configuration properties it is provided with to determine their validity. If the configuration properties are considered valid, this method should return silently. If the configuration properties are considered invalid, this method should throw an `InvalidConfigurationException` with an appropriate message.

- The `raVerifyConfiguration` method must be declared as `public` and cannot be `static` or `abstract`.
- The implementation of this method can assume that at least the `setResourceAdaptorContext` method has been invoked on the resource adaptor object, but must otherwise assume nothing about the internal state of the resource adaptor object. In particular, the implementation of this method should not depend on the state of any instance variables in the resource adaptor object other than those that are initialized by the resource adaptor object's constructor or `setResourceAdaptorContext` method.
- The resource adaptor object must not keep a reference to the `ConfigProperties` object provided as an argument to this method after the method returns. The `ConfigProperties` object is owned by the SLEE.

This method takes the following argument:

- The `properties` argument is a `javax.slee.resource.ConfigProperties` object that contains the set of configuration properties for the resource adaptor entity associated with the resource adaptor object. For more information on `ConfigProperties` object refer to section 14.12.1.

This method throws the following exception:

- `javax.slee.resource.InvalidConfigurationException`.
  The method throws an `InvalidConfigurationException` if any of the configuration properties specified in the `properties` argument are not valid for the Resource Adaptor.

This method is invoked by the SLEE in response to the following methods being invoked by the Administrator on a `ResourceManagementMBean` object:

- The `createResourceAdaptorEntity` method.
  When the `createResourceAdaptorEntity` method is invoked on a `ResourceManagementMBean` object, the SLEE invokes the `raVerifyConfiguration` method on a resource adaptor object in the Unconfigured state.
  - If the `raVerifyConfiguration` method returns without throwing an exception, the SLEE assumes that the configuration properties specified by the Administrator are valid for the Resource Adaptor. The SLEE then creates zero or more additional resource adaptor objects as required and transitions each of them to the Inactive state, passing the same set of configuration properties to each resource adaptor object's `raConfigure` method as was passed to the `raVerifyConfiguration` method.
  - If the `raVerifyConfiguration` method returns by throwing an exception, the SLEE assumes that the configuration properties specified by the Administrator are not valid for the Resource Adaptor. The `createResourceAdaptorEntity` management method subsequently returns by throwing an `InvalidConfigurationException`.
- The `updateConfigurationProperties` method.
  When the `updateConfigurationProperties` method is invoked on a `ResourceManagementMBean` object, the SLEE invokes the `raVerifyConfiguration` method on a resource adaptor object in any valid lifecycle state.
  - If the `raVerifyConfiguration` method returns without throwing an exception, the SLEE assumes that the configuration properties specified by the Administrator are valid for the Resource Adaptor. The SLEE then invokes the `raConfigurationUpdate` method on each resource adaptor object of the resource adaptor entity that is in the Inactive, Active, or Stopping state, passing the same set of configuration properties to each `raConfigurationUpdate` method as was passed to the `raVerifyConfiguration` method.
  - If the `raVerifyConfiguration` method returns by throwing an exception, the SLEE assumes that the configuration properties specified by the Administrator are not valid for the Resource Adaptor. The `updateConfigurationProperties` management method returns by throwing an `InvalidConfigurationException`.

15.12.2.2 `ResourceAdaptor` interface `raConfigurationUpdate` method

This method is invoked by the SLEE to notify the resource adaptor object that the Administrator has modified the configuration properties of the associated resource adaptor entity.

Before this method is invoked, the `raVerifyConfiguration` method is invoked on a resource adaptor object of the same resource adaptor entity to validate the configuration properties supplied by the Administrator. If the `raVerifyConfiguration` method returns without throwing an exception, the SLEE assumes that the configuration properties are valid for the Resource Adaptor and proceeds to invoke the `raConfigurationUpdate` method with the same set of configuration properties in order to reconfigure the resource adaptor object.

If the value of the `supports-active-reconfiguration` attribute of the `resource-adaptor-class` element in the resource adaptor entity's Resource Adaptor deployment descriptor is "False", the `raConfigurationUpdate` method will only be invoked on a resource adaptor object in the Inactive state. If the `supports-active-reconfiguration` attribute is set to "True", the `raConfigurationUpdate` method may be called on a resource adaptor object that is in the Inactive, Active, or Stopping state.

- The `raConfigurationUpdate` method must be declared as `public` and cannot be `static` or `abstract`.

This method takes the following argument:

- The `properties` argument specifies the new configuration for the resource adaptor entity associated with the resource adaptor object. This argument specifies the current value for all configuration properties for the resource adaptor entity. The resource adaptor object is responsible for inspecting this argument to determine any appropriate internal action necessary to realize the requested configuration. For more information on `ConfigProperties` object refer to section 14.12.1.

This method throws no checked exceptions. However, if this method returns by throwing an unchecked exception, the SLEE must log the exception. Regardless of whether or not this method returns by throwing an unchecked exception, the resource adaptor object invoked is considered by the SLEE to have been reconfigured.

After this method returns, the SLEE must not modify the `ConfigProperties` object passed to the resource adaptor object as the argument to this method, as ownership of the `ConfigProperties` object is transferred to the resource adaptor object as a result of this method invocation.

15.12.3 Interface Access methods

These methods allow the SLEE to configure its own runtime environment, and the environment of any SBBs that use the resource adaptor entity of the resource adaptor object.

These methods may be called on a resource adaptor object in the Inactive, Active, or Stopping state.

15.12.3.1 `ResourceAdaptor` interface `getResourceAdaptorInterface` method

The SLEE invokes this method to get access to an object that implements the resource adaptor interface of a resource adaptor type implemented by the Resource Adaptor. The resource adaptor interface object enables SBBs to invoke the resource adaptor entity, for example to create calls or send messages.

- The resource adaptor interface is specified as part of the resource adaptor type[40]. As a Resource Adaptor can implement more than one resource adaptor type, the class name of the resource adaptor interface required by the SLEE is passed as an argument to this method.
- The class of the returned object must be assignable to the resource adaptor interface class named by the `className` argument.
- The returned object can be invoked at any time while the resource adaptor object is in the Inactive, Active, or Stopping state. It is expected that the implementation of the resource adaptor interface may have to reject invocations if the resource adaptor object is not in the appropriate state to fulfill the invocation. For example, it is not typically valid to send a message to the network when the resource adaptor object is in the Inactive state.
- The resource adaptor class must return a non-null object for each resource adaptor type that it implements which defines a resource adaptor interface. The SLEE will not invoke this method for resource adaptor types that do not define a resource adaptor interface.

The method takes the following argument:

- The `className` argument is the fully qualified class name of the resource adaptor type's resource adaptor interface. Resource adaptors can implement one or more resource adaptor types. This argument determines which resource adaptor interface is required by the SLEE.

[40] For example in JSR 32 (JAIN SIP API Specification) the resource adaptor interface is the `javax.sip.SipProvider` interface (see Appendix D).

15.12.3.2 `ResourceAdaptor` interface `getMarshaler` method

The SLEE invokes this method to get a reference to a Resource Adaptor's Marshaler object. A Marshaler object implements the `javax.slee.resource.Marshaler` interface (see Section 15.15). A Marshaler object is used by the SLEE to convert between object and distributable forms of both Events and Activity Handles[41].

This method takes no input parameters. If the Resource Adaptor does not implement the `Marshaler` interface this method returns the value `null`. If the Resource Adaptor returns `null` from this method, the SLEE is unable to marshal Events or Activity Handles for the Resource Adaptor, and consequently the Resource Adaptor must not set the `SLEE_MAY_MARSHAL` flag when starting an Activity or firing an Event to the SLEE.

15.12.4 Event filter methods

The event filter methods enable a Resource Adaptor to perform optimizations which filter out "unnecessary" events. In order to perform such optimizations the Resource Adaptor is told which event types may be received by Services installed in the SLEE, and therefore may avoid firing events which are not of those event types. Events may be received by Services which are in the Active or Stopping states which have an event handler method for the event type. In the case that the Resource Adaptor fires events for which there are no interested Services, the SLEE will still process the events, however doing so may use compute resource when it is not necessary to do so.

Implementing event filtering logic in the Resource Adaptor class is optional. A Resource Adaptor class must implement the event filter methods, but may do so with an empty code body if it does not care to perform any action when these methods are invoked by the SLEE.

The event filter methods are coarse grained, that is to say they are defined based on the lifecycle of each Service and the event types which may be received by that Service. The methods are called once for each Service as it enters the appropriate state.

Event filter methods are only invoked on a resource adaptor object for a particular Service if there is at least one event type received by that Service that the Resource Adaptor may fire to the SLEE, as determined by the resource adaptor types implemented by the Resource Adaptor. In addition, the list of event types included in the argument to the event filter methods only includes event types referenced by the resource adaptor types implemented by the Resource Adaptor.

For example, consider the deployment scenario illustrated in Figure 24.

- When Service A changes state:
  - An event filter method will be invoked on resource adaptor objects of Resource Adaptor A, and the list of event types passed as a parameter to the event filter method will be { Event Type A, Event Type B }.
  - An event filter method will be invoked on resource adaptor objects of Resource Adaptor B, and the list of event types passed as a parameter to the event filter method will be { Event Type C }.
- When Service B changes state:
  - No event filter method will be invoked on resource adaptor objects of Resource Adaptor A as Service B does not receive any events from the resource adaptor types implemented by Resource Adaptor A.
  - An event filter method will be invoked on resource adaptor objects of Resource Adaptor B, and the list of event types passed as a parameter to the event filter method will be { Event Type C }.

[41] The Marshaler allows SLEE implementations to "move" Event and Activity Handle objects through multiple JVMs via marshaling and unmarshaling.

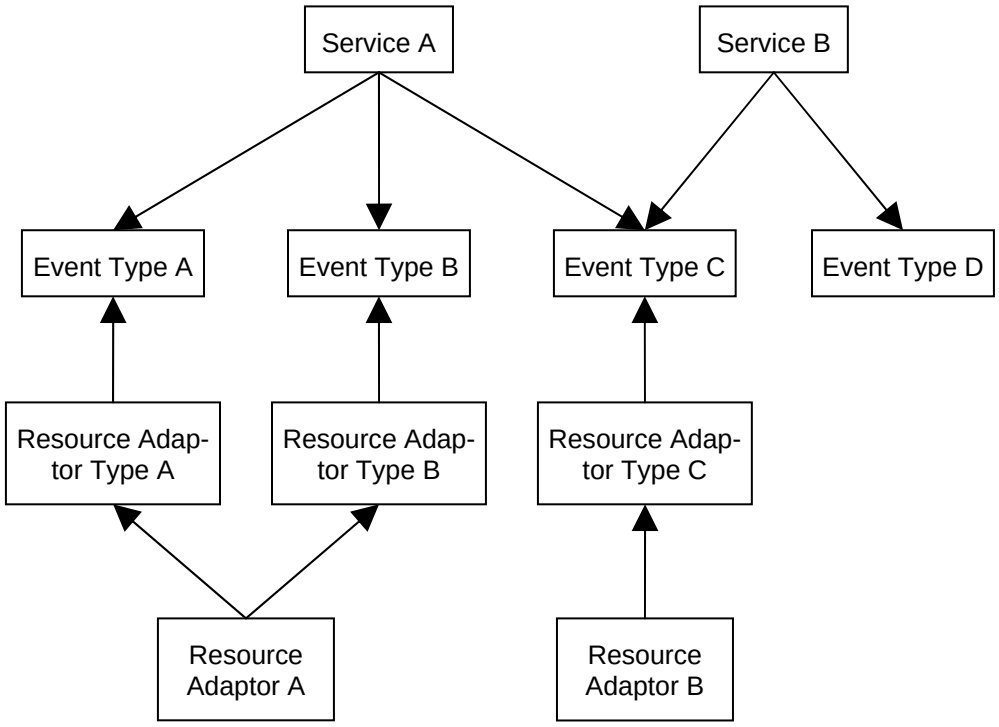

Figure 24 An example deployment scenario. Arrows indicate component dependency references, or in the case of Services, references to event types received by the Service.

If a Resource Adaptor has specified to ignore event type checking (see Section 15.10) then the rules above are relaxed and the SLEE will invoke the event filter methods on resource adaptor objects for that Resource Adaptor whenever a state change occurs in any Service. In additional all event types that can be received by the Service are included in the event filter method callback argument.

The argument to each event filter method is a `ReceivableService` object (see Section 13.8.1). A `ReceivableService` object contains the service component identifier and an array of `ReceivableService.ReceivableEvent` objects (see Section 13.8.2). The service component identifier identifies the Service that has changed state in the SLEE. The array of `ReceivableService.ReceivableEvent` objects identify the event types that can be received by the Service. If there are multiple SBBs in a particular Service that each receive the same event type but specify different `event-resource-option` values in their deployment descriptor for the event type, then when an event filter method is invoked on a resource adaptor object for that Service a `ReceivableService.ReceivableEvent` object for the same event type will be appear multiple times in the array. Each occurrence of the event type will include a unique event resource option for the SBBs in the Service. It is the responsibility of the Resource Adaptor to resolve these conflicts. For example, in JCC, one SBB in a Service may request a particular JCC event in Notify mode, while another SBB in the same Service may request the same JCC event in Blocking mode. A JCC Resource Adaptor may resolve this by using the Blocking mode for the Service for that event type, in preference to the Notify mode.

The event filter methods throw no checked exceptions. However, if an event filter method returns by throwing an unchecked exception, the SLEE must log the exception.

15.12.4.1 `ResourceAdaptor` interface `serviceActive` method

The SLEE invokes this method on a resource adaptor object to inform it that a Service that may receive events of one or more event types fired by the resource adaptor object has transitioned to the Active state. The Resource Adaptor is expected to update any internal state related to event filters as required.

This method takes the following argument:

- The `serviceInfo` argument is a `ReceivableService` object that identifies the service that has entered the Active state and the type of events that the Service has event handlers for.

15.12.4.2 `ResourceAdaptor` interface `serviceStopping` method

The SLEE invokes this method on a resource adaptor object to inform it that a Service that may receive events of one or more event types fired by the resource adaptor object has transitioned to the Stopping state. The Resource Adaptor is expected to update any internal state related to event filters as required.

This method takes the following argument:

- The `serviceInfo` argument is a `ReceivableService` object that identifies the service that has entered the Stopping state and the type of events that the Service has event handlers for.

15.12.4.3 `ResourceAdaptor` interface `serviceInactive` method

The SLEE invokes this method on a resource adaptor object to inform it that a Service that may receive events of one or more event types fired by the resource adaptor object has transitioned from the Stopping state to the Inactive state. The Resource Adaptor is expected to update any internal state related to event filters as required.

This method takes the following argument:

- The `serviceInfo` argument is a `ReceivableService` object that identifies the service that has entered the Inactive state and the type of events that the Service has event handlers for.

15.12.5 Mandatory callback methods

The mandatory callback methods are invoked by the SLEE. These methods may be invoked by the SLEE in any thread, therefore the resource adaptor object should be written to handle both concurrent and re-entrant invocations of these methods. As an example a re-entrant invocation may occur if a Resource Adaptor invokes a method on the `SleeEndpoint` interface and the SLEE invokes a mandatory callback method in the same thread before returning.

The methods `queryLiveness`, `administrativeRemove`, and `getActivity` are invoked on the resource adaptor object that started the Activity.

15.12.5.1 `ResourceAdaptor` interface `queryLiveness` method

The SLEE invokes this method to query if a specific Activity belonging to this resource adaptor object is still valid. This implies that the Resource Adaptor will check the underlying resource to see if the Activity is still active. For example, it may be possible that ending of an Activity did not occur due to message loss in the SLEE. In this case the SLEE would retain the Activity Context of the Activity and SBBs attached to the Activity Context. If the Activity is not alive the Resource Adaptor is expected to end the activity (via the `SleeEndpoint` interface). If the Activity is still alive the Resource Adaptor is not expected to do anything. The method takes the following argument:

- The `handle` argument is the Activity Handle of the Activity that the SLEE is querying.

The resource adaptor object should not block in this method. If the resource adaptor object needs to query an external system to determine if the Activity is “alive”, it should send an asynchronous query or perform the query in a separate thread, rather than blocking the invoking thread while awaiting a response from the external system.

This method is not intended to be used for the normal ending of an activity. Rather it is intended to be used where an Activity is present in the SLEE and is suspected to be no longer valid.

15.12.5.2 `ResourceAdaptor` interface `getActivity` method

The SLEE invokes this method to get access to the Activity object for an Activity Handle. The Resource Adaptor is expected to return a non-`null` object.

The method takes the following argument:

- The `handle` argument specifies the Activity Handle of the requested Activity.

This method may be invoked by the SLEE within a transaction. If the method needs to access transacted state to obtain the activity object then it should either use the enclosing transaction to do so (if invoked within a transaction), or start a new transaction (if invoked without a transaction). An example scenario is where a Resource Adaptor uses transacted storage (such as Profiles, or a database) for activity state.

15.12.5.3 `ResourceAdaptor` interface `getActivityHandle` method

The SLEE invokes this method to obtain an Activity Handle for an Activity object. This method may be invoked by the SLEE when the SLEE needs to construct or look-up an Activity Context for the Activity. For example, the implementation in the SLEE of a resource adaptor type's Activity Context Interface Factory would require the Activity Handle of the Activity object passed in to the factory in order to construct the Activity Context for the Activity object.

- A resource adaptor entity must be able to distinguish its own Activity objects from those of other resource adaptor entities created from the same Resource Adaptor. For example, if a Resource Adaptor called *FooResource* has two resource adaptor entities *FooA* and *FooB*, then *FooA* must be able to distinguish between Activity objects it created and Activity objects created by *FooB*[42].
- If the Resource Adaptor receives this invocation, owns the Activity object, but has not yet started the Activity in the SLEE, then before this method returns the Resource Adaptor must start the Activity using the `SleeEndpoint` interface. This technique allows Resource Adaptors to implement lazy-creation of activities in the case where Activity objects of different types are related in some way. For example, a JCC Resource Adaptor may only start the `JccConnection` Activities as it fires `JccConnectionEvent` Events, and only start the `JccCall` Activity associated with a `JccConnection` Activity if the `getActivityHandle` method is invoked for the `JccCall` Activity. The motivation for this is that many SBBs may only be interested in the `JccConnection` Activities and the `JccConnectionEvent` Events, so there is no point in these cases for the JCC Resource Adaptor to create unnecessary Activities in the SLEE. Rather, the `JccCall` Activities are created "on-demand".
- This method must return `null` if the supplied Activity object was not created by the resource adaptor entity. This means that the SLEE can use this method to determine which resource adaptor entity of a specific resource adaptor type "owns" the activity. This "probing" is necessary in the implementation of an Activity Context Interface Factory as there may be several resource adaptor entities in the SLEE which implement the resource adaptor type that defined the Activity Context Interface Factory, and hence several possible sources of the Activity object.

The method takes the following argument:

- The `activity` argument is the Activity object for which the Activity Handle is being requested.

An example implementation of a `getActivityHandle` method that fulfills the contract described above is shown below:

```
public ActivityHandle getActivityHandle(Object activity) {
        try {
```

[42] A convenient approach is to include a unique identifier, such as the resource adaptor entity's name or a reference to the resource adaptor object, in the Activity object's state, which is checked by the resource adaptor object.

```
                MyActivityImpl m = (MyActivityImpl)activity;
                // the activity object stores a reference to some object
                // that is specific to the resource adaptor entity
                if (m.getOwner() == this) {
                        // the activity object was created by this resource adaptor entity
                        //
                        // this activity object includes a ref to its activity handle
                        // so return that handle
                        return m.getActivityHandle();
                }
        }
        catch (ClassCastException e) { /* fallthrough */ }

        // not my activity
        return null;
}
```

The `getActivityHandle` method may be invoked by the SLEE within a transaction. If the method needs to access transacted state to obtain the activity handle then it should either use the enclosing transaction to do so (if invoked within a transaction), or start a new transaction (if invoked without a transaction). An example scenario is where a Resource Adaptor uses transacted storage (such as Profiles, or a database) for activity state.

15.12.5.4 `ResourceAdaptor` interface `administrativeRemove` method

The SLEE invokes this method in order to inform the resource adaptor object that an Activity it started has been removed from the SLEE due to an administrative action. The Resource Adaptor is expected to remove any internal state related to the Activity. Additionally the Resource Adaptor may perform a protocol level operation to clean up any protocol-peer related state.

The method takes the following argument:

- The handle argument is the Activity Handle of the Activity that was removed from the SLEE.

15.12.6 Optional callback methods

The optional callback methods are invoked by the SLEE if and only if the Resource Adaptor requested callbacks. The Resource Adaptor requests callbacks via using the flags parameter of various methods on the `SleeEndpoint` interface. These methods may be invoked by the SLEE in any thread, therefore the resource adaptor object should be written to handle both concurrent and re-entrant invocations of these methods.

The `eventProcessingSuccessful`, `eventProcessingFailed`, and `eventUnreferenced` methods are invoked on the resource adaptor object which fired the particular event. The `activityEnded` and `activityUnreferenced` methods are invoked on the resource adaptor object that owns the activity.

15.12.6.1 `ResourceAdaptor` interface `eventProcessingSuccessful` method

The SLEE invokes this method to inform the resource adaptor object that the specified Event was processed successfully by the SLEE. An Event is considered to be processed successfully if the SLEE has attempted to deliver the Event to all interested SBBs. It is not required that the SBB event handler or SBB rolled back transactions commit for event processing to still be successful, unless failure to commit is caused by a system-level failure.

This method is only invoked if the Event was fired with the `EventFlags.REQUEST_EVENT_PROCESSING_SUCCESSFUL_CALLBACK` flag set in the `eventFlags` parameter of the fire-event method invoked on the `SleeEndpoint` interface.

This method takes the following arguments:

- The `handle` argument is the Activity Handle of the Activity upon which the event was fired.
- The `eventType` argument specifies the event type of the Event fired.
- The `event` argument is the Event object that was fired on the Activity.
- The `address` argument is the address that was supplied by the Resource Adaptor when the event was fired. If a `null` address was specified, this value is also `null`.
- The `service` argument is the `ReceivableService` object that was supplied by the Resource Adaptor when the event was fired. If a `null` `ReceivableService` object was specified, this value is also `null`.
- The `flags` argument includes information that enables any post-event processing required by the Resource Adaptor. The following flags may be included in this value:
    - `EventFlags.SBB_PROCESSED_EVENT` (see Section 15.18)

15.12.6.2 `ResourceAdaptor` interface `eventProcessingFailed` method

The SLEE invokes this method to inform the resource adaptor object that the specified Event could not be processed successfully by the SLEE. Event processing can fail if, for example, the SLEE has insufficient resource to process the Event, a SLEE node fails during event processing or a system level failure prevents the SLEE from committing transactions.

This method is only invoked if the Event was fired with the `EventFlags.REQUEST_EVENT_PROCESSING_FAILED_CALLBACK` flag set in the `eventFlags` parameter of the fire-event method invoked on the `SleeEndpoint` interface.

This method takes the following arguments:

- The `handle` argument is the Activity Handle of the Activity upon which the event was fired.
- The `eventType` argument specifies the event type of the Event fired.
- The `event` argument is the Event object that was fired on the Activity.
- The `address` argument is the address that was supplied by the Resource Adaptor when the Event was fired. If a `null` address was specified, this value is also `null`.
- The `service` argument is the `ReceivableService` object that was supplied by the Resource Adaptor when the event was fired. If a `null` `ReceivableService` object was specified, this value is also `null`.
- The `flags` argument includes information that enables any post-event processing required by the Resource Adaptor. The following flags may be included in this value:
    - `EventFlags.SBB_PROCESSED_EVENT` (see Section 15.18)
- The `reason` argument informs the Resource Adaptor why the event processing failed (see section 15.16).

15.12.6.3 `ResourceAdaptor` interface `eventUnreferenced` method

The SLEE invokes this method to inform the Resource Adaptor that the SLEE no longer references an Event object which was previously fired by the resource adaptor object. The SLEE may continue to reference an Event object after the `eventProcessingSuccessful` or `eventProcessingFailed` callback for the Event if, for example, an SBB refires the Event on another Activity. This method allows the resource adaptor object to pool Event objects.

The SLEE no longer references an Event object when the following conditions are true:

- There are no more SBB entities to deliver the Event to.
- Any Event Context which references the Event is not suspended.

- Any event processing transactions for the Event have committed or rolled back.
- Any callbacks related to the Event (other than this method) have been invoked on the resource adaptor object.

Note: an SBB object may still hold a reference to the Event object using Java language references. Such a practice is an application programming error, and the SLEE and Resource Adaptors need not concern themselves with these situations.

This method is only invoked if the Event was fired with the `EventFlags.REQUEST_EVENT_UNREFERENCED_CALLBACK` flag set in the `eventFlags` parameter of the fire-event method invoked on the `SleeEndpoint` interface.

This method takes the following arguments:

- The handle argument is the Activity Handle of the Activity upon which the event was fired
- The `eventType` argument specifies the event type of the Event fired.
- The event argument is the Event object that was fired on the Activity.
- The address argument is the address that was supplied by the Resource Adaptor when the Event was fired. If a null address was specified, this value is also null.
- The `service` argument is the `ReceivableService` object that was supplied by the Resource Adaptor when the event was fired. If a `null` `ReceivableService` object was specified, this value is also `null`.
- The `flags` argument is the necessary information that enables any post-event processing required by the resource adaptor. There are currently no SLEE specification defined flags that are passed to this method, however a SLEE implementation may pass vendor-defined flags if desired.

15.12.6.4 `ResourceAdaptor` interface `activityEnded` method

The SLEE invokes this method to inform the Resource Adaptor that the SLEE has completed activity end processing for the Activity represented by the Activity Handle parameter. The Resource Adaptor should release any resources related to this Activity as the SLEE will not ask for it again.

This method is only invoked if the Activity was started with the `ActivityFlags.REQUEST_ACTIVITY_ENDED_CALLBACK` flag set.

This method takes the following argument:

- The `handle` argument is the Activity Handle of the Activity that has ended.

15.12.6.5 `ResourceAdaptor` interface `activityUnreferenced` method

The SLEE invokes this method to inform the Resource Adaptor that the Activity Context of the specified Activity Handle has no attached SBB entities, is no longer referenced by any SLEE Facilities, and has no Events being processed or waiting to be processed on the Activity. The resource adaptor can choose to end the Activity at this time if desired[43]. Note that this method is purely a notification to the Resource Adaptor. The SLEE itself will perform no further action on an Activity Context that is unreferenced. In particular the SLEE will not automatically end the Activity of an unreferenced Activity Context. If the Resource Adaptor wants the Activity to end when it becomes unreferenced, then it is the responsibility of the Resource Adaptor to explicitly end the activity using the `SleeEndpoint` interface when this callback method is invoked.

This method is only invoked if the Activity was started with the `ActivityFlags.REQUEST_ACTIVITY_UNREFERENCED_CALLBACK` bit set.

[43] If the Resource Adaptor ends the activity, then the Resource Adaptor can offer the same "Garbage Collection" approach used by the SLEE to implicitly end Null Activities.

The method takes the following argument:

- The `handle` argument is the Activity Handle of the Activity that is unreferenced according to the prescribed conditions.

15.13 `ResourceAdaptorContext` interface

The `ResourceAdaptorContext` interface is implemented by the SLEE. It provides the Resource Adaptor with the required capabilities in the SLEE to execute its work. The `ResourceAdaptorContext` object holds references to a number of objects that are of interest to many Resource Adaptors. A resource adaptor object is provided with a `ResourceAdaptorContext` object when the `setResourceAdaptorContext` method of the `ResourceAdaptor` interface is invoked on the resource adaptor object.

The `ResourceAdaptorContext` interface is defined as follows:

```
package javax.slee.resource;

import java.util.Timer;
import javax.slee.ServiceID;
import javax.slee.resource.ResourceAdaptorID;
import javax.slee.resource.ResourceAdaptorTypeID;
import javax.slee.SLEEException;
import javax.slee.facilities.AlarmFacility;
import javax.slee.facilities.EventLookupFacility;
import javax.slee.facilities.ServiceLookupFacility;
import javax.slee.facilities.Tracer;
import javax.slee.profile.ProfileTable;
import javax.slee.profile.UnrecognizedProfileTableNameException;
import javax.slee.transaction.SleeTransactionManager;
import javax.slee.usage.NoUsageParametersInterfaceDefinedException;
import javax.slee.usage.UnrecognizedUsageParameterSetNameException;

public interface ResourceAdaptorContext {
        public ResourceAdaptorID getResourceAdaptor();
        public ResourceAdaptorTypeID[] getResourceAdaptorTypes();
        public String getEntityName();
        public SleeEndpoint getSleeEndpoint();
        public AlarmFacility getAlarmFacility();
        public Tracer getTracer(String tracerName)
                throws  NullPointerException,
                        IllegalArgumentException,
                        SLEEException;
        public EventLookupFacility getEventLookupFacility();
        public ServiceLookupFacility getServiceLookupFacility();
        public SleeTransactionManager getSleeTransactionManager();
        public Timer getTimer();
        public ProfileTable getProfileTable(String profileTableName)
                throws  NullPointerException,
                        UnrecognizedProfileTableNameException,
                        SLEEException;
        public Object getDefaultUsageParameterSet()
                throws  NoUsageParametersInterfaceDefinedException,
                        SLEEException;
        public Object getUsageParameterSet(String paramSetName)
                throws  NullPointerException,
                        NoUsageParametersInterfaceDefinedException,
                        UnrecognizedUsageParameterSetNameException,
                        SLEEException;
        public ServiceID getInvokingService();
}
```

All methods on the `ResourceAdaptorContext` interface are non-transactional.

- The `getResourceAdaptor` method.
  This method returns the component identifier of the resource adaptor.

- The `getResourceAdaptorTypes` method.
  This method returns the component identifiers for the resource adaptor types implemented by the resource adaptor. The returned array is always non- null.
- The `getEntityName` method.
  This method returns the name of the resource adaptor entity the resource adaptor object was created for.
- The `getSleeEndpoint` method.
  This method returns a `SleeEndpoint` object that the resource adaptor object may use to notify the SLEE about Activities that are starting and ending, and to fire Events on those Activities to the SLEE.
- The `getAlarmFacility` method.
  This method returns an Alarm Facility object that can be used to raise, update and clear alarms in the SLEE. The notification source used by this Alarm Facility is a `javax.slee.management.ResourceAdaptorEntityNotification` which contains the name of the resource adaptor entity as identified by the `getEntityName` method. Alarm notifications generated by this Alarm Facility are of the type `javax.slee.management.ResourceAdaptorEntityNotification.ALARM_NOTIFICATION_TYPE`. For more information on alarms and the alarm facility refer to section 13.2.
- The `getTracer` method.
  This method returns a `Tracer` object for the tracer named by the `tracerName` argument. The notification source used by the tracer is a `javax.slee.management.ResourceAdaptorEntityNotification` which contains the name of the resource adaptor entity as identified by the `getEntityName` method. Refer to the Trace Facility in Section 13.3 for a complete discussion on tracers and tracer names. Trace notifications generated by a tracer obtained using this method are of the type `javax.slee.management.ResourceAdaptorEntityNotification.TRACE_NOTIFICATION_TYPE`.

  This method throws the following exceptions:
  - A `NullPointerException` is thrown if `tracerName` is null.
  - An `IllegalArgumentException` is thrown if tracerName is an invalid name. Name components within a tracer name must have at least one character. For example “com.mycompany” is a valid tracer name, whereas “com..mycompany” is not.
  - A `SLEEException` is thrown if a `Tracer` object could not be obtained due to a system-level failure.
- The `getEventLookupFacility` method.
  This method returns an Event Lookup Facility object. The Event Lookup Facility is used to convert from event type component identifiers to the `FireableEventType` objects used by Resource Adaptors (see Section 15.9).
- The `getServiceLookupFacility` method.
  This method returns a Service Lookup Facility object. The Service Lookup Facility is used to obtain information about the types of events that a Service may receive.
- The `getSleeTransactionManager` method.
  This method returns a SLEE Transaction Manager object that can be used to demarcate transaction boundaries and access the current transaction (if any).
- The `getTimer` method.
  This method returns a shared `java.util.Timer` object that can be used by Resource Adaptors. The `cancel` method of the `Timer` object returned may not be invoked by Resource Adaptors. Attempting to do so will throw a `java.lang.UnsupportedOperationException`.

- The `getProfileTable` method.
  This method returns a `ProfileTable` object for the Profile Table specified by the `profileTableName` argument. The object returned by this method may be safely typecast to the Profile Table Interface defined by the profile specification of the profile table. However, if the Resource Adaptor has not declared a `profile-spec-ref` in its deployment descriptor for the Profile Specification of the Profile Table specified by the `profileTableName` argument, then the Resource Adaptor may not have the necessary classes in its classloader to perform this typecast. In addition, for the same reasons, the Resource Adaptor may not be able to interact with Profile Local Objects obtained from the returned `ProfileTable` object other than via the generic SLEE-defined `ProfileLocalInterface`.

  This method throws the following exceptions:

  - A `NullPointerException` is thrown if the `profileTableName` argument is `null`.
  - An `UnrecognizedProfileTableNameException` is thrown if no profile table exists with the specified name.
  - A `SLEEException` is thrown if a `ProfileTable` object could not be obtained due to a system-level failure.

- The `getDefaultUsageParameterSet` method.
  This method returns the unnamed usage parameter set for the resource adaptor entity. The fully scoped name used to locate the usage parameter set consists of the name of the resource adaptor entity as identified by the `getEntityName` method and the undefined "name" of the unnamed parameter set. The object returned by this method may be safely typecast to the Usage Parameters Interface defined by the Resource Adaptor.
  This method throws the following exceptions:

  - A `NoUsageParametersInterfaceDefinedException` is thrown if the Resource Adaptor has not defined a Usage Parameters Interface.
  - A `SleeException` is thrown if the usage parameter set could not be returned due to a system-level failure.

- The `getUsageParameterSet` method.
  This method returns a named usage parameter set for the resource adaptor entity. The fully scoped name used to locate the usage parameter set consists of the name of the resource adaptor entity as identified by the `getEntityName` method and the name specified by the `paramSetName` argument. The object returned by this method may be safely typecast to the Usage Parameters Interface defined by the Resource Adaptor.
  This method throws the following exceptions:

  - A `NullPointerException` is thrown if the `paramSetName` argument is `null`.
  - A `NoUsageParametersInterfaceDefinedException` is thrown if the Resource Adaptor has not defined a Usage Parameters Interface.
  - An `UnrecognizedUsageParameterSetNameException` is thrown if the `paramSetName` argument does not identify a named usage parameter set created by the Administrator for the resource adaptor entity.
  - A `SleeException` is thrown if the usage parameter set could not be returned due to a system-level failure.

- The `getInvokingService` method.
  This method returns the Service component identifier of the Service currently being invoked by the SLEE in the current thread. The purpose of this method is to allow Resource Adaptors to determine which Service an SBB belongs to when that SBB invokes a downcall on the Resource Adaptor, for example by invoking a method on the Resource Adaptor interface or on an Activity

object. This method returns `null` if there is no Service currently being invoked by the SLEE in the current thread.

15.14 `SleeEndpoint` interface

The SLEE Endpoint defines the interface used by a Resource Adaptor to start and end Activities within the SLEE, fire Events on those Activities, and to suspend an Activity. A SLEE must implement the `SLEEEndpoint` interface to be both multi-thread safe and reentrant safe.

The `SleeEndpoint` interface is defined as follows:

```
package javax.slee.resource;

import javax.slee.Address;
import javax.slee.ServiceID;
import javax.slee.TransactionRequiredLocalException;
import javax.slee.UnrecognizedServiceException;
import javax.slee.SLEEException;

public interface SleeEndpoint {
        // start-activity methods
        public void startActivity(ActivityHandle handle, Object activity)
                throws  NullPointerException,
                        IllegalStateException,
                        ActivityAlreadyExistsException,
                        StartActivityException,
                        SLEEException;
        public void startActivity(
                        ActivityHandle handle, Object activity, int activityFlags)
                throws  NullPointerException,
                        IllegalStateException,
                        ActivityAlreadyExistsException,
                        StartActivityException,
                        SLEEException;
        public void startActivitySuspended(ActivityHandle handle, Object activity)
                throws  NullPointerException,
                        IllegalStateException,
                        TransactionRequiredLocalException,
                        ActivityAlreadyExistsException,
                        StartActivityException,
                        SLEEException;
        public void startActivitySuspended(
                        ActivityHandle handle, Object activity, int activityFlags)
                throws  NullPointerException,
                        IllegalStateException,
                        TransactionRequiredLocalException,
                        ActivityAlreadyExistsException,
                        StartActivityException,
                        SLEEException;
        public void startActivityTransacted(ActivityHandle handle, Object activity)
                throws  NullPointerException,
                        IllegalStateException,
                        TransactionRequiredLocalException,
                        ActivityAlreadyExistsException,
                        StartActivityException,
                        SLEEException;
        public void startActivityTransacted(
                        ActivityHandle handle, Object activity, int activityFlags)
                throws  NullPointerException,
                        IllegalStateException,
                        TransactionRequiredLocalException,
                        ActivityAlreadyExistsException,
                        StartActivityException,
                        SLEEException;


        // end-activity methods
        public void endActivity(ActivityHandle handle)
```

```
                throws  NullPointerException,
                        UnrecognizedActivityHandleException,
                        SLEEException;
        public void endActivityTransacted(ActivityHandle handle)
                throws  NullPointerException,
                        TransactionRequiredLocalException,
                        UnrecognizedActivityHandleException,
                        SLEEException;

        // event submission methods
        public void fireEvent(ActivityHandle handle, FireableEventType eventType,
                              Object event, Address address, ReceivableService service)
                throws  NullPointerException,
                        UnrecognizedActivityHandleException,
                        IllegalEventException,
                        ActivityIsEndingException,
                        FireEventException,
                        SLEEException;
        public void fireEvent(ActivityHandle handle, FireableEventType eventType,
                              Object event, Address address, ReceivableService service,
                              int eventFlags)
                throws  NullPointerException,
                        UnrecognizedActivityHandleException,
                        IllegalEventException,
                        ActivityIsEndingException,
                        FireEventException,
                        SLEEException;
        public void fireEventTransacted(ActivityHandle handle,
                              FireableEventType eventType, Object event, Address address,
                              ReceivableService service)
                throws  NullPointerException,
                        UnrecognizedActivityHandleException,
                        IllegalEventException,
                        TransactionRequiredLocalException,
                        ActivityIsEndingException,
                        FireEventException,
                        SLEEException;
        public void fireEventTransacted(ActivityHandle handle,
                              FireableEventType eventType, Object event, Address address,
                              ReceivableService service, int eventFlags)
                throws  NullPointerException,
                        UnrecognizedActivityHandleException,
                        IllegalEventException
                        TransactionRequiredLocalException,
                        ActivityIsEndingException,
                        FireEventException,
                        SLEEException;

        // miscellaneous methods
        public void suspendActivity(ActivityHandle handle)
                throws  NullPointerException,
                        UnrecognizedActivityHandleException,
                        TransactionRequiredLocalException,
                        SLEEException;

}
```

The SLEE Endpoint provides several methods to start Activities. These methods vary in that the processing state of the started Activity is different and whether or not the start of an Activity is part of an enclosing transaction. The methods are overloaded to allow flags, indicating whether or not the callback methods related to the lifecycle of the Activity should be invoked by the SLEE, to be optionally specified. The methods defined for starting Activities are as follows:

- The `startActivity` methods.
  These methods provide Resource Adaptors with a non-transacted mechanism to tell the SLEE that there is a new Activity and provide to the SLEE an Activity Handle that represents the Activity.

  - This method is non-transactional method. This also implies that the Activity does not get enrolled in any transaction.
  - The SLEE considers the Activity started regardless of whether the transaction associated with the current thread (if it exists) commits or rolls back.
  - These methods are typically used by resources such as protocol stacks which are non-transacted. For example, a non-transaction resource may create a connection. This connection is created and exists regardless of whether the current transaction commits or rolls back.

- The `startActivityTransacted` methods.
  These methods provide Resources Adaptors with a transacted mechanism to tell the SLEE that there is a new Activity and provide to the SLEE an Activity Handle that represents the Activity.
  - This method is a mandatory transactional method.
  - This Activity is visible only within the current transaction (i.e. the transaction associated with the current thread) until the transaction commits successfully.
  - If the transaction commits, then the SLEE should "remember" the new Activity and the Activity becomes visible to subsequent transactions as well as non-transacted operations.
  - If the transaction rolls back, then the SLEE should "forget" about the new Activity (i.e. roll back its knowledge of the new Activity) and the Activity cannot be visible to subsequent transactions (as the SLEE no longer knows that it exists).
  - These methods are typically used by transacted resources. An implementation of a Null Activity Factory may use these methods to inform the SLEE of newly created Null Activities.

The SLEE Endpoint provides two methods to end Activities. Both methods cause the Activity Context of the Activity to transition into the Ending state, however one operation is part of an enclosing transaction whilst the other is not. These methods defined for ending Activities are:

- The `endActivity` method.
  This method provides a resource adaptor with a basic mechanism for ending an Activity in the SLEE.
  - This is a non-transactional method.
  - This method is typically used by resources such as protocol stacks which are non-transacted.

- The `endActivityTransacted` method.
  This method provides a resource adaptor with a transacted mechanism to end an Activity.
  - This is a mandatory transactional method.
  - If a particular transaction ends an Activity using this method and that transaction commits, the transition of the Activity's Activity Context to the Ending state becomes permanent and visible to subsequent transactions. If a transaction which ends an Activity using this method does not commit, the transition of the Activity Context to the Ending state is rolled back and subsequent transactions continue to view the Activity Context of the Activity as Active. When an Activity is ended using this method, the transition of that Activity's Activity Context to the Ending state is visible only to the ending transaction until that transaction commits successfully. Once the ending transaction has committed successfully the Activity end is visible to other transactions, and is visible to non-transacted operations.
  - This method is typically used by transacted resources such as the Null Activity Factory in the way that Null Activities are explicitly ended in the SLEE.

The SLEE Endpoint provides several methods to fire Events. These methods allow the Resource Adaptor to specify whether or not an Event is fired as part of an enclosing transaction. The methods are overloaded to

specify whether or not the SLEE should deliver call back methods to the Resource Adaptor based on the outcome of processing the Event. The methods defined for firing Events are:

- The `fireEvent` methods.
  These methods provide Resource Adaptors with a basic mechanism for firing Events on Activities.
  - These are non-transactional methods.
  - These methods are typically used by resources such as protocol stacks which are non-transacted.
- The `fireEventTransacted` methods.
  These methods provide a Resource Adaptor with a transacted mechanism to fire Events on Activities.
  - These are mandatory transactional methods.
  - If a particular transaction fires an Event using these methods and that transaction commits, the Event is accepted for processing by the SLEE. If a transaction which fires an Event using this method does not commit, the Event is discarded by the SLEE as if it was never fired.
  - These methods are typically used by transacted resources that wish to fire events in a transacted manner. Events fired by SBBs can also be considered to have been fired by a transacted resource.

The `suspendActivity` method.
This method is used to suspend an activity. Each suspension of an Activity inserts a logical "event processing barrier" into the SLEEs internal processing state for an Activity. Events fired post the insertion of an event processing barrier are not processed until the barrier is removed. Events fired prior to the insertion of an event processing barrier may be delivered. A barrier is removed once the suspending transaction has completed (committed or rolled back). There may be multiple barriers related to an Activity at a given point in time, with events fired prior to and post each barrier.

- This method is a mandatory transactional method. Unlike typical transactional methods, it does not result in transactional state changes in the objects involved, i.e. it does change the state of the Activity or ActivityContext. Its only side effect is on transaction itself, i.e. by suspending the the Activity in the transaction.
- This method may be used to avoid concurrency conflicts that may occur if optimistic concurrency control is in use[44]. A common example is when a response to an outgoing request arrives at the SLEE *before* the transaction in which the request was generated completes. A more detailed scenario is described (assuming optimistic concurrency control):
  An SBB wants to attach itself to a non-transacted new Activity in order to receive events fired on the Activity. For example, if an SBB invokes a Resource Adaptor to create an Activity. This Activity may represent a query. The Resource Adaptor may send a request message to an external system as part of creating the Activity. When the Resource Adaptor receives a response, it will fire an event that represents the response on the Activity. A typical SBB invocation sequence for this example would be as follows:
  a. SBB entity invokes a Resource Adaptor to create an Activity that represents the query. The Resource Adaptor also sends the request and returns an Activity object to the SBB.
  b. SBB entity invokes an Activity Context Interface Factory to obtain the Activity Context associated with Activity object.
  c. SBB entity attaches to the Activity Context because the SBB entity wants to receive the event representing the response.

[44] If a Resource Adaptor does not use suspendActivity appropriately then appropriate pessimistic locking in the SLEE is sufficient to avoid concurrency conflicts.

If the Resource Adaptor receives the response before step (c), then it is possible that it could fire the event on the Activity and the SLEE could attempt to deliver the event to SBB entities attached to the Activity Context associated with the Activity. Since the SBB entity has not attached itself to the Activity Context, the SLEE will not deliver the response event to the SBB entity and the SBB entity will not have a chance to process the response. In summary, due to the use of optimistic locking there is race between the SBB completing step (c) and the SLEE delivering of the response event. This occurs because the non-transacted new Activity does not suspend the new Activity in the transaction. On the other hand, if Resource Adaptor had suspended the new Activity in the transaction, then the SLEE would not be able to initiate any event delivery prior to the transaction completing. Therefore this method enables optimistic concurrency control to function in common request/response scenarios without concurrency conflicts.

15.14.1 Guidelines for using starting Activities and suspending new Activities

The following are best practice guidelines for how Resource Adaptors should use the start Activity and suspend Activity methods.

Situation	Resource Adaptor Guideline
Transacted resource invoked to create Activity, e.g. SBB creates a `NullActivity`.	1. Perform operation on the resource to have it start the Activity. 2. Allocate an Activity Handle. 3. Invoke `startActivityTransacted` on SLEE Endpoint.
Non-transacted resource invoked to create Activity within a transaction, e.g. an SBB invokes a non-transacted resource that represents a network protocol stack.	1. Perform operation on the resource to have it start the Activity. 2. Allocate an Activity Handle. 3. Invoke `startActivitySuspended` on SLEE Endpoint.
Non-transacted resource invoked to create Activity without a transaction, e.g. a network protocol stack invokes a Resource Adaptor to handle a new incoming request and Resource Adaptor has to start a new Activity to process the incoming request.	1. Perform operation on the resource to have it start the Activity (often implied by receipt of a message from the protocol stack). 2. Allocate an Activity Handle. 3. Invoke `startActivity` on SLEE Endpoint.

15.14.2 `SleeEndpoint` interface `startActivity` methods

The `startActivity` methods are used by the resource adaptor entity to inform the SLEE that the resource adaptor entity has created a new Activity. These methods are non-transactional.

These methods should be used by Resource Adaptors that start activities in a non-transactional manner (such as Resource Adaptors for non-transactional protocol stacks) in response to events generated by the network or non-transactional API calls by SBBs.

The `startActivity` method is overloaded. The first form of the method takes the following argument:

- The `handle` argument specifies the Activity Handle of the Activity that has been started by the resource adaptor entity.
- The `activity` argument specifies the Activity object of the Activity.

The second form of the method takes the following arguments:

- The `handle` argument specifies the Activity Handle of the Activity that has been started by the resource adaptor entity.
- The `activity` argument specifies the Activity object of the Activity.
- The `activityFlags` argument identifies the SLEE features which the resource adaptor entity wants to make use of for the new Activity. The supported flags are specified by the `ActivityFlags` class as described in Section 15.17.

Invoking the first form of the method is semantically identical to invoking the second form of the method with the `activityFlags` argument equal to `ActivityFlags.NO_FLAGS`.

These methods throws the following exceptions:

- The `NullPointerException` is thrown if a `null` Activity Handle, or Activity object is passed as an argument.
- The `IllegalStateException` is thrown if the invoking resource adaptor object is in the Inactive or Stopping state.
- The `ActivityAlreadyExistsException` is thrown if the Activity Handle identifies an Activity that already exists in the SLEE.
- The `StartActivityException` is thrown if the Activity could not be started due to a non system level issue, such as input rate-limiting.
- The `SLEEException` is thrown if the Activity could not be started in the SLEE due to a system-level failure.

15.14.3 `SleeEndpoint` interface `startActivityTransacted` methods

The `startActivityTransacted` methods are used by the resource adaptor entity to inform the SLEE that the resource adaptor entity has created a new Activity within a SLEE transaction. If this method returns without throwing an exception then the Activity is started within the context of the enclosing transaction only. The Activity is only visible outside of the enclosing transaction after the enclosing transaction commits. If the enclosing transaction rolls back, the SLEE views the Activity as if it was never started.

Whilst the enclosing transaction that starts an Activity is still active, the Activity is only visible within the context of the enclosing transaction. Therefore Events may only be fired on that Activity in a transacted manner using the `fireEventTransacted` methods. Using a `fireEvent` method to attempt to fire an Event in an untransacted manner on such an Activity whilst the enclosing transaction has not completed causes the `fireEvent` method to throw an `UnrecognizedActivityHandleException`, as the Activity is not yet visible outside of the enclosing transaction. After the enclosing transaction has committed, Events may be fired on the Activity in either a transacted or non-transacted manner as appropriate. Similarly, an Activity started in a transacted manner can only be ended in the same enclosing transaction if it is also ended in a transacted manner using the `endActivityTransacted` method.

These methods are mandatory transactional. A resource adaptor entity may obtain a valid transaction context through transaction context propagation from an SBB entity invoking a method on an interface or class defined by a resource adaptor type, or via the `SleeTransactionManager` interface (see Section 9.14.2).

The `startActivityTransacted` method is overloaded. The first form of the method takes the following argument:

- The `handle` argument specifies the Activity Handle of the Activity that has been started by the resource adaptor entity.
- The `activity` argument specifies the Activity object of the Activity.

The second form of the method takes the following arguments:

- The `handle` argument specifies the Activity Handle of the Activity that has been started by the resource adaptor entity.
- The `activity` argument specifies the Activity object of the Activity.
- The `activityFlags` argument identifies the SLEE features which the resource adaptor entity wants to make use of for the new Activity. The supported flags are specified by the `ActivityFlags` class as described in Section 15.17.

Invoking the first form of the method is semantically identical to invoking the second form of the method with the `activityFlags` argument equal to `ActivityFlags.NO_FLAGS`.

These methods throws the following exceptions:

- The `NullPointerException` is thrown if a `null` Activity Handle, or Activity object is passed as an argument.
- The `IllegalStateException` is thrown if the invoking resource adaptor object is in the Inactive or Stopping state.
- The `TransactionRequiredLocalException` is thrown if the calling thread does not contain a valid transaction context.
- The `ActivityAlreadyExistsException` is thrown if the Activity Handle identifies an Activity that already exists in the SLEE.
- The `StartActivityException` is thrown if the Activity could not be started due to a non system level issue, such as input rate-limiting.
- The `SLEEException` is thrown if the Activity could not be started in the SLEE due to a system-level failure.

15.14.4 `SleeEndpoint` interface `startActivitySuspended` methods

The `startActivitySuspended` methods perform the function of the equivalent `startActivity` method[45] and `suspendActivity` method but combined into a single atomic action.

These methods should be used by Resource Adaptors that start activities in a non-transactional manner in response to non-transactional API calls by SBBs, however they have the same mandatory transactional semantics as the `suspendActivity` method. Refer to the description of the `suspendActivity` method (Section 15.14.5) for more information.

The `startActivitySuspended` method is overloaded. The first form of the method takes the following argument:

- The `handle` argument specifies the Activity Handle of the Activity that has been started by the resource adaptor entity.
- The `activity` argument specifies the Activity object of the Activity.

The second form of the method takes the following arguments:

- The `handle` argument specifies the Activity Handle of the Activity that has been started by the resource adaptor entity.
- The `activity` argument specifies the Activity object of the Activity.
- The `activityFlags` argument identifies the SLEE features which the resource adaptor entity wants to make use of for the new Activity. The supported flags are specified by the `ActivityFlags` class as described in Section 15.17.

[45] "Equivalent" meaning the `startActivity` method with the same list of arguments as the invoked `startActivitySuspended` method.

Invoking the first form of the method is semantically identical to invoking the second form of the method with the `activityFlags` argument equal to `ActivityFlags.NO_FLAGS`.

These methods throws the following exceptions:

- The `NullPointerException` is thrown if a `null` Activity Handle, or Activity object is passed as an argument.
- The `IllegalStateException` is thrown if the invoking resource adaptor object is in the Inactive or Stopping state.
- The `TransactionRequiredLocalException` is thrown if the calling thread does not contain a valid transaction context.
- The `ActivityAlreadyExistsException` is thrown if the Activity Handle identifies an Activity that already exists in the SLEE.
- The `StartActivityException` is thrown if the Activity could not be started due to a non system level issue, such as input rate-limiting.
- The `SLEEException` is thrown if the Activity could not be started in the SLEE due to a system-level failure.

15.14.5 `SleeEndpoint` interface `suspendActivity` method

The `suspendActivity` method is used by the resource adaptor entity to suspend an activity. It may be used by the resource adaptor to avoid concurrency conflicts if optimistic locking is in use. For more information refer to the overview text in section 15.14.

Each suspension of an Activity inserts a logical "event processing barrier" into the SLEEs internal processing state for an Activity. Events fired post the insertion of an event processing barrier are not processed until the barrier is removed. Events fired prior to the insertion of an event processing barrier may be delivered. A barrier is removed once the suspending transaction has completed (committed or rolled back). There may be multiple barriers related to an Activity at a given point in time, with events fired prior to and post each barrier.

This method is mandatory transactional, however it does not read or write transacted state. A resource adaptor entity may obtain a valid transaction context through transaction context propagation from an SBB entity invoking a method on an interface or class defined by a resource adaptor type. Alternatively, the resource adaptor entity may obtain a valid transaction context using the `SleeTransactionManager` interface (see Section 9.14.2).

These methods throws the following exceptions:

- The `NullPointerException` is thrown if a `null` Activity Handle object is passed as an argument.
- The `UnrecognizedActivityHandleException` is thrown if the Activity Handle object passed as an argument does not represent an Activity that exists in the SLEE.
- The `TransactionRequiredLocalException` is thrown if the calling thread does not contain a valid transaction context.
- The `SLEEException` is thrown if the Activity could not be started in the SLEE due to a system-level failure or other reason such as input rate limiting.

15.14.6 `SleeEndpoint` interface `endActivity` method

This method is used by the resource adaptor entity to inform the SLEE that an Activity has ended. This method is non-transactional. When this method returns the Activity Context for the specified Activity will have entered the Ending state and an Activity End Event will have been queued for processing on the Activity. Further events may not be fired on the Activity, either by Resource Adaptors or SBBs.

This method should be used by Resource Adaptors that start and end activities in a non-transactional manner (such as Resource Adaptors for non-transactional protocol stacks) in response to events generated by the network or non-transactional API calls by SBBs.

This method has no further effect if the Activity Context of the specified Activity is already in the Ending state in the SLEE.

This method takes the following argument:

- The `handle` argument specifies the Activity Handle of the Activity that has ended.

This method throws the following exceptions:

- The `NullPointerException` is thrown if a `null` Activity Handle object is passed as an argument.
- The `UnrecognizedActivityHandleException` is thrown if the Activity Handle object passed as an argument does not represent an Activity that exists in the SLEE.
- The `SLEEException` is thrown if the Activity could not be ended in the SLEE due to a system-level failure.

15.14.7 `SleeEndpoint` interface `endActivityTransacted` method

This method is used by the resource adaptor entity to inform the SLEE that an Activity has ended within a SLEE transaction. When this method returns the Activity Context for the specified Activity will have entered the Ending state within the context of the enclosing transaction, and an Activity End Event will have been queued on the Activity for processing if the transaction commits. Further events may not be fired on the Activity within the context of the same transaction, either by Resource Adaptors or SBBs. If the enclosing transaction commits, the transition of the Activity Context to the Ending state will be considered permanent and subsequent transactions will also view the Activity Context as being in the Ending state. The Activity will end in the SLEE after the SLEE processes the Activity End Event. If the enclosing transaction rolls back, then the transacted transition of the Activity Context to the Ending state will be rolled back also and the Activity End Event will be discarded by the SLEE.

This method is mandatory transactional. A resource adaptor entity may obtain a valid transaction context through transaction context propagation from an SBB entity invoking a method on an interface or class defined by a resource adaptor type, or via the `SleeTransactionManager` interface (see Section 9.14.2).

This method has no further effect if the Activity Context of the specified Activity is already in the Ending state in the SLEE.

This method takes the following argument:

- The `handle` argument specifies the Activity Handle of the Activity that has ended.

This method throws the following exceptions:

- The `NullPointerException` is thrown if a `null` Activity Handle object is passed as an argument.
- The `TransactionRequiredLocalException` is thrown if the calling thread does not contain a valid transaction context.
- The `UnrecognizedActivityHandleException` is thrown if the Activity Handle object passed as an argument does not represent an Activity that exists in the SLEE.
- The `SLEEException` is thrown if the Activity could not be ended in the SLEE due to a system-level failure.

15.14.8 `SleeEndpoint` interface `fireEvent` methods

These methods are used by resource adaptor entities to fire Events on Activities in the SLEE. These methods are non-transactional. The action of firing an Event is free from any transactional semantics, more specifically the Event is fired and can be accepted by the SLEE for processing regardless of the outcome of any enclosing transaction.

This method should be used by Resource Adaptors that start and end Activities in a non-transactional manner (such as Resource Adaptors for non-transactional protocol stacks) in response to events generated by the network or non-transactional API calls by SBBs.

The ordering of Events fired via this method is self-consistent. Events fired on the same Activity by a Resource Adaptor using this method from a single thread will be processed by the SLEE in the order that they were fired. However, no consistency guarantees are made between Events fired from different threads, Events fired on different Activities, or between an Event fired in a non-transacted manner and an Event fired in a transacted manner, where both Events are fired on the same Activity within the same transaction context.

The `fireEvent` method is overloaded. The first form of this method takes the following arguments:

- The `handle` argument specifies the Activity Handle of the Activity that the Event is being fired on.
- The `eventType` argument specifies the event type of the Event being fired. `FireableEventType` objects can be obtained from the Event Lookup Facility (see Section 13.7).
- The `event` argument is the Event being fired on the Activity. This argument cannot be `null`.
- The `address` argument is the default address on which this Event is to be fired. If no default address exists this value may be `null`.
- The `service` argument is an optional argument. If this argument is not `null` then only SBB entities belonging to the specified Service are eligible to receive the event and only new root SBB entities belonging to the specified Service are eligible to be initiated by the event. If this argument is `null`, SBB entities belonging to all Services are eligible to receive the event and new root SBB entities belonging to all Services are eligible to be initiated by the event. (See Section 8.6 for more details on event routing and other eligibility requirements.) `ReceivableService` objects may be obtained from the Service Lookup Facility (see Section 13.8) or from the event filter methods invoked on the Resource Adaptor class (see Section 15.12.4)

The second form of this method takes the same arguments as the first method and adds the following additional argument:

- The `eventFlags` argument identifies the SLEE features which the resource adaptor entity wants to make use of for the Event. For example, the resource adaptor entity may request to receive callbacks when event processing is complete in order to determine if the Event was processed successfully. The supported flags are specified by the `EventFlags` class as described in Section 15.18.

These methods throw the following exceptions:

- The `NullPointerException` is thrown if a `null` Activity Handle or Event object is passed as an argument.
- The `UnrecognizedActivityHandleException` is thrown if the Activity Handle object passed as an argument does not represent an Activity that exists in the SLEE.
- The `IllegalEventException` is thrown if any of the following conditions occur:
    - The `eventType` argument is not a `FireableEventType` object generated by the SLEE and provided to the Resource Adaptor via an Event Lookup Facility.
    - The resource adaptor attempts to fire an event of an event type it is not permitted to fire. A resource adaptor object may only fire events of event types referenced by the resource

adaptor types implemented by the Resource Adaptor of the resource adaptor object unless this restriction has been disabled (see Section 15.10).

- The class of the Event object fired is not assignable to the event class of the event type as identified by `eventType` argument.

- The `ActivityIsEndingException` is thrown if the Activity Context of the specified Activity is in the Ending state. Events may not be fired on Activities that are ending.
- The `FireEventException` is thrown if the SLEE could not accept the event for processing due to a non system level issue, such as input rate-limiting.
- The `SLEEException` is thrown if the Event could not be accepted by the SLEE due to a system-level failure.

15.14.9 `SleeEndpoint` interface `fireEventTransacted` methods

These methods are used by resource adaptor entities to fire Events on activities in the SLEE within a transaction context. An event fired using one of these methods is only accepted for processing by the SLEE if the enclosing transaction commits. If the transaction rolls back, the SLEE considers the event to not have been fired.

This method should be used by Resource Adaptors that start and end activities in a transactional manner. For example, an implementation of a Resource Adaptor that generates Null Activities on behalf of the SLEE would use this method to fire Events on a Null Activity.

The ordering of Events fired via this method is self-consistent. Events fired on the same Activity by a Resource Adaptor using this method from a single thread will be processed by the SLEE in the order that they were fired. However, no consistency guarantees are made between Events fired from different threads, Events fired on different Activities, or between an Event fired in a non-transacted manner and an Event fired in a transacted manner, where both Events are fired on the same Activity within the same transaction context.

These methods are mandatory transactional. A resource adaptor entity may obtain a valid transaction context through transaction context propagation from an SBB entity invoking a method on an interface or class defined by a resource adaptor type, or via the `SleeTransactionManager` interface (see Section 9.14.2).

The `fireEventTransacted` method is overloaded. The first form of this method takes the following arguments:

- The `handle` argument specifies the Activity Handle of the Activity that the Event is being fired on.
- The `eventType` argument specifies the event type of the Event being fired. `FireableEventType` objects can be obtained from the Event Lookup Facility (see Section 13.7).
- The `event` argument is the Event being fired on the Activity. This argument cannot be `null`.
- The `address` argument is the default address on which this Event is to be fired. If no default address exists this value may be `null`.
- The `service` argument is an optional argument. If this argument is not `null` then only SBB entities belonging to the specified Service are eligible to receive the event and only new root SBB entities belonging to the specified Service are eligible to be initiated by the event. If this argument is `null`, SBB entities belonging to all Services are eligible to receive the event and new root SBB entities belonging to all Services are eligible to be initiated by the event. (See Section 8.6 for more details on event routing and other eligibility requirements.) `ReceivableService` objects may be obtained from the Service Lookup Facility (see Section 13.8) or from the event filter methods invoked on the Resource Adaptor class (see Section 15.12.4)

The second form of this method takes the same arguments as the first method and adds the following additional argument:

- The `eventFlags` argument identifies the SLEE features which the resource adaptor entity wants to make use of for the Event. For example, the resource adaptor entity may request to receive callbacks when event processing is complete in order to determine if the Event was processed successfully. The supported flags are specified by the `EventFlags` class as described in Section 15.18.

These methods throw the following exceptions:

- The `NullPointerException` is thrown if a `null` Activity Handle or Event object is passed as an argument
- The `UnrecognizedActivityHandleException` is thrown if the Activity Handle object passed as an argument does not represent an Activity that exists in the SLEE.
- The `IllegalEventException` is thrown if any of the following conditions occur:
  - The `eventType` argument is not a `FireableEventType` object generated by the SLEE and provided to the Resource Adaptor via an Event Lookup Facility.
  - The resource adaptor attempts to fire an event of an event type it is not permitted to fire. A resource adaptor object may only fire events of event types referenced by the resource adaptor types implemented by the Resource Adaptor of the resource adaptor object unless this restriction has been disabled (see Section 15.10).
  - The class of the Event object fired is not assignable to the event class of the event type as identified by `eventType` argument.
- The `TransactionRequiredLocalException` is thrown if the calling thread does not contain a valid transaction context.
- The `ActivityIsEndingException` is thrown if the Activity Context of the specified Activity is in the Ending state. Events may not be fired on Activities that are ending.
- The `FireEventException` is thrown if the SLEE could not accept the event for processing due to a non system level issue, such as input rate-limiting.
- The `SLEEException` is thrown if the Event could not be accepted by the SLEE due to a system-level failure or other reason such as input rate limiting.

15.15 `Marshaler` interface

Resource Adaptors may optionally provide an implementation of the `Marshaler` interface. A Marshaler is used by the SLEE to convert Events and Activity Handles between object and marshaled forms. If a Resource Adaptor does not provide an implementation of the `Marshaler` interface then the Resource Adaptor is not permitted to use the `SLEE_MAY_MARSHAL` flag of the `ActivityFlags` and `EventFlags` class when starting Activities or firing Events.

The `Marshaler` interface is as follows:

```
package javax.slee.resource;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import java.nio.ByteBuffer;

public interface Marshaler {
       public int getEstimatedEventSize(FireableEventType eventType, Object event);
       public ByteBuffer getEventBuffer(FireableEventType eventType, Object event);
       public void releaseEventBuffer(FireableEventType eventType, Object event,
                              ByteBuffer buffer);
       public void marshalEvent(FireableEventType eventType, Object event,
```

```
                                DataOutput out)
            throws IOException;
        public Object unmarshalEvent(FireableEventType eventType, DataInput in)
            throws IOException;

        public int getEstimatedHandleSize(ActivityHandle handle);
        public void marshalHandle(ActivityHandle handle, DataOutput out)
            throws IOException;
        public ActivityHandle unmarshalHandle(DataInput in)
            throws IOException;
}
```

- The `getEstimatedEventSize` method.
  This method is invoked by the SLEE to get an estimation of the size (in bytes) of the marshaled form of an Event object generated by the Resource Adaptor. It can be used by the SLEE to help size appropriately any internal buffers used to store the marshaled form of an Event.
  The `eventType` argument specifies the event type of the Event. The `event` argument specifies the Event object for which the SLEE is requesting a size estimate. The `event` argument may be `null`. If the `event` argument is null, the SLEE is asking the Resource Adaptor for a general estimate of the marshaled size of all Event objects of the event type specified by the `eventType` argument. If the `event` argument is not `null`, the SLEE is asking the Resource Adaptor for an estimate of the specified Event object's marshaled size.
  A SLEE implementation must correctly handle the case where the estimate returned from this method is too low, and the act of marshaling an object exceeds the resource originally allocated. This method is intended only to help avoid resizing marshaling resources in the common case.
- The `getEventBuffer` method.
  This method is invoked by the SLEE to request a `ByteBuffer` containing an already marshaled form of the Event object specified by the `event` argument. The `eventType` argument specifies the event type of the Event.
  The exact purpose of this method is to give the Resource Adaptor an opportunity to provide the SLEE with a view buffer onto an existing buffer that already contains the marshaled form of the Event object. For example, the Resource Adaptor may cache network buffers that contain the original marshaled form of the event received off the wire. If the Resource Adaptor can provide the SLEE with such a buffer, the SLEE and Resource Adaptor do not have to expend work to marshal the Event object again unnecessarily.
  If a Resource Adaptor does not have such a view buffer available to give to the SLEE, the Resource Adaptor should return `null` from this method. In this case, the SLEE falls back to using the `marshalEvent` method to obtain a marshaled form of the Event object.
  If a Resource Adaptor returns a non-`null` `ByteBuffer` object from this method, the Resource Adaptor must ensure that the `ByteBuffer` object remains valid until a corresponding `releaseEventBuffer` method invocation is received from the SLEE for that buffer.
- The `releaseEventBuffer` method.
  This method is invoked by the SLEE when it no longer requires a `ByteBuffer` object previously obtained by using the Marshaler object's `getEventBuffer` method. The `eventType` and `event` arguments to this method are the same arguments that were provided to the `getEventBuffer` method invocation that returned the `ByteBuffer` specified by the `buffer` argument.
  A Resource Adaptor object is free to release any resources associated with the specified `ByteBuffer` when this method is invoked as the SLEE will no longer use the buffer.
- The `marshalEvent` method.
  This method is invoked by the SLEE when it needs to obtain the marshaled form of an Event object. The Resource Adaptor should write a serialized form of the Event object specified by the `event` argument using the `DataOutput` object specified by the `out` argument. The `event`-

`Type` argument identifies the event type of the Event object.
This method may throw the following exception:

- `java.io.IOException`.
  This exception is thrown if the Event object could not be successfully marshaled.

- The un`marshalEvent` method.
  This method is invoked by the SLEE when it needs to unmarshal an Event object previously marshaled by the Resource Adaptor. The Resource Adaptor should read the marshaled form of the Event object from the `DataInput` object specified by the `in` argument and reconstruct the Event object. The `eventType` argument identifies the event type of the Event object that is marshaled and provides access to the class name of the event type and a class loader that can be used to load the event class if required.
  This method may throw the following exceptions:

  - `java.io.IOException`.
    This exception is thrown if the Event object could not be successfully unmarshaled.

- The `getEstimatedHandleSize` method.
  This method is invoked by the SLEE to get an estimation of the size (in bytes) of the marshaled form of an Activity Handle object generated by the Resource Adaptor. It can be used by the SLEE to help size appropriately any internal buffers used to store the marshaled form of an Activity Handle.
  The `handle` argument specifies the `ActivityHandle` object for which the SLEE is requesting a size estimate. The `handle` argument may be `null`. If the `handle` argument is null, the SLEE is asking the Resource Adaptor for a general estimate of the marshaled size of all Activity Handle objects. If the `handle` argument is not `null`, the SLEE is asking the Resource Adaptor for an estimate of the specified `ActivityHandle` object's marshaled size.
  A SLEE implementation must correctly handle the case where the estimate returned from this method is too low, and the act of marshaling an object exceeds the resource originally allocated. This method is intended only to help avoid resizing marshaling resources in the common case.

- The `marshalHandle` method.
  This method is invoked by the SLEE when it needs to obtain the marshaled form of an `ActivityHandle` object. The Resource Adaptor should write a serialized form of the `ActivityHandle` object specified by the `handle` argument using the `DataOutput` object specified by the `out` argument.
  This method may throw the following exception:

  - `java.io.IOException`.
    This exception is thrown if the Activity Handle could not be successfully marshaled.

- The un`marshalHandle` method.
  This method is invoked by the SLEE when it needs to unmarshal an `ActivityHandle` object previously marshaled by the Resource Adaptor. The Resource Adaptor should read the marshaled form of the Activity Handle from the `DataInput` object specified by the `in` argument and reconstruct the `ActivityHandle` object.
  This method may throw the following exceptions:

  - `java.io.IOException`.
    This exception is thrown if the Activity Handle could not be unsuccessfully unmarshaled.

15.16 FailureReason class

The `FailureReason` class defines an enumerated type for event processing failures. A `FailureReason` object informs a Resource Adaptor the reason why event processing failed in the SLEE for a particular event.

The public interface of the `FailureReason` class is as follows.

```
package javax.slee.resource;

public class FailureReason {
        // singletons
        public static final FailureReason OTHER_REASON …;
        public static final FailureReason EVENT_QUEUE_FULL …;
        public static final FailureReason EVENT_QUEUE_TIMEOUT …;
        public static final FailureReason SYSTEM_OVERLOAD …;
        public static final FailureReason EVENT_MARSHALING_ERROR …;
        public static final FailureReason FIRING_TRANSACTION_ROLLED_BACK …;

        // integer representation
        public static final int REASON_OTHER_REASON …;
        public static final int REASON_EVENT_QUEUE_FULL …;
        public static final int REASON_EVENT_QUEUE_TIMEOUT …;
        public static final int REASON_SYSTEM_OVERLOAD …;
        public static final int REASON_EVENT_MARSHALING_ERROR …;
        public static final int REASON_FIRING_TRANSACTION_ROLLED_BACK …;

        // constructor
        protected FailureReason(int reason) { … }

        // methods
        public boolean isOtherReason() { … }
        public boolean isEventQueueFull() { … }
        public boolean isEventQueueTimeout() { … }
        public boolean isSystemOverload() { … }
        public boolean isEventMarshalingError() { … }
        public boolean isFiringTransactionRolledBack() { … }

        public static FailureReason fromInt(int reason)
                throws IllegalArgumentException { … }
        public int toInt() { … }

        public int hashCode() { … }
        public boolean equals(Object o) { … }
        public String toString() { … }
}
```

- A singleton instance of each SLEE-defined enumerated value is guaranteed (via an implementation of the `readResolve` method, see `java.io.Serializable` for more details), so that equality tests using == are always evaluated correctly. For convenience, an integer representation of `FailureReason` objects is also available.
- Each of the `is<`*`Failure`*`>` methods determines if this `FailureReason` object represents the <*FAILURE*> of the event failure, and is equivalent to `(this == <`*`FAILURE`*`>)`. For example, the `isSystemOverload` method determines if this `FailureReason` object represents the SYSTEM_OVERLOAD reason as to why the event processing failed and is equivalent to `(this == SYSTEM_OVERLOAD)`.
- The `fromInt` and `toInt` methods allow conversion between the `FailureReason` object form and numeric form.

A SLEE vendor may define any implementation-specific reason codes that may be useful or appropriate by extending this class. The integer representation of implementation-specific event codes must be positive numbers greater than zero. The `REASON_OTHER_REASON` failure reason code has the reserved value 0, while negative numbers are reserved for the SLEE-specification defined reason codes defined in this class.

In order to support vendor extensions to this class, the following methods have non-typical behavior in the `FailureReason` class:

- The `isOtherReason` method.
  This method returns `true` if `this` is equal to the `FailureReason.OTHER_REASON` object or if the integer representation of the failure reason code is greater than zero, i.e. a vendor-defined reason.

- The `fromInt` method.
  If the reason code specified by the `reason` argument is equal to or less than zero and identifies a valid SLEE-defined reason code, the appropriate singleton `FailureReason` object is returned. If the specified reason code is equal to or less than zero but does not identify a SLEE-defined reason code, the method throws an `IllegalArgumentException`. If the specified reason code is greater than zero, a `FailureReason` object whose `toInt` method returns the same code is returned. This allows a valid `FailureReason` object to be constructed outside the SLEE from the integer reason code form when the SLEE vendor's `FailureReason` subclass is unknown or not available.
- The `toInt` method.
  This method returns either a SLEE-defined failure reason code or a vendor-defined failure reason code.

The SLEE specification makes no guarantees regarding the singleton instance property of the `FailureReason` enumerated type for vendor-defined reason codes.

15.17 ActivityFlags class

The `ActivityFlags` class defines a set of flags that relate to Activities. These flags are used in the methods on the `SleeEndpoint` interface which start new Activities in the SLEE. The flags identify SLEE functions that a Resource Adaptor may make use of with respect to the Activity the flags are combined with.

The public interface of the ActivityFlags class is as follows:

```
package javax.slee.resource;

public final class ActivityFlags {
        // constants
        public static final int NO_FLAGS …;
        public static final int STANDARD_FLAGS_MASK …;
        public static final int VENDOR_FLAGS_MASK …;

        // flags
        public static final int SLEE_MAY_MARSHAL …;
        public static final int REQUEST_ACTIVITY_ENDED_CALLBACK …;
        public static final int REQUEST_ACTIVITY_UNREFERENCED_CALLBACK …;

        // methods to test flags
        public static boolean hasNoFlags(int flags) { … }
        public static boolean hasStandardFlags(int flags) { … }
        public static boolean hasVendorFlags(int flags) { … }
        public static boolean hasFlags(int flags, int flagsToTestFor) { … }
        public static boolean hasSleeMayMarshal(int flags) { … }
        public static boolean hasRequestActivityEndedCallback(int flags) { … }
        public static boolean hasRequestActivityUnreferencedCallback(int flags) { … }

        // methods to set flags
        public static int setSleeMayMarshal(int currentFlags) { … }
        public static int setRequestActivityEndedCallback(int currentFlags) { … }
        public static int setRequestActivityUnreferencedCallback(int currentFlags) { … }

        public static String toString(int flags) { … }
}
```

The following constants are defined:

- `NO_FLAGS`.
  If a flags parameter is set to this value it indicates that there are no flags set.
- `STANDARD_FLAGS_MASK`.
  This constant defines a bit-mask that covers the range for current and future SLEE specification defined flag values. Bits 0-23 are reserved for use for the SLEE specification.

- VENDOR_FLAGS_MASK.
  This constant defines a bit-mask that covers the range for vendor extension flag values. Bits 24-31 are reserved for use by SLEE vendors.

The following flags are defined:

- The SLEE_MAY_MARSHAL flag.
  If the SLEE_MAY_MARSHAL flag is set when an Activity is started then the SLEE implementation may, at its discretion, use the Resource Adaptor's Marshaler object to marshal and unmarshal the Activity Handle for the Activity. Refer to Section 15.15 for a description of the `Marshaler` interface implemented by the Marshaler object.
- The REQUEST_ACTIVITY_ENDED_CALLBACK flag.
  If this flag is set when an Activity is started then the SLEE implementation must invoke the `activityEnded` callback method on the resource adaptor object when the Activity has ended in the SLEE. If this flag is not set then the SLEE implementation will not invoke the `activityEnded` callback method for that Activity. For more information refer to the description of the `activityEnded` callback method in Section 15.12.6.4.
- The REQUEST_ACTIVITY_UNREFERENCED_CALLBACK flag.
  If this flag is set when the Activity is started then the SLEE implementation must invoke the `activityUnreferenced` callback method on the resource adaptor object if the Activity becomes unreferenced in the SLEE. If this flag is not set then the SLEE implementation will not invoke the `activityUnreferenced` callback method for that Activity. For more information refer to the description of the `activityUnreferenced` callback method in Section 15.12.6.5.

The following methods are defined:

- The `hasNoFlags` method.
  This method returns `true` if the supplied `flags` parameter is equal to the constant NO_FLAGS.
- The `hasStandardFlags` method.
  This method returns `true` if the supplied `flags` parameter has one or more flags set which are in the STANDARD_FLAGS_MASK range (i.e. one or more of bits 0-23 are set).
- The `hasVendorFlags` method.
  This method returns `true` if the supplied `flags` parameter has one or more flags set which are in the VENDOR_FLAGS_MASK range (i.e. one or more of bits 24-31 are set).
- The `hasFlags` method.
  This method returns `true` if all the flags indicated by the `flagsToTestFor` parameter are set in the `flags` parameter.
- The `hasSleeMayMarshal` method.
  This method returns `true` if the supplied `flags` parameter has the SLEE_MAY_MARSHAL flag set.
- The `hasRequestActivityEndedCallback` method.
  This method returns `true` if the supplied `flags` parameter has the REQUEST_ACTIVITY_ENDED_CALLBACK flag set.
- The `hasRequestActivityUnreferencedCallback` method.
  This method returns `true` if the supplied `flags` parameter has the REQUEST_ACTIVITY_UNREFERENCED_CALLBACK flag set.
- The `setSleeMayMarshal` method.
  This method returns the result of setting the SLEE_MAY_MARSHAL flag in the supplied `currentFlags` parameter. This method uses a bitwise OR to set the SLEE_MAY_MARSHAL flag.

- The `setRequestActivityEndedCallback` method.
  This method returns the result of setting the `REQUEST_ACTIVITY_ENDED_CALLBACK` flag in the supplied `currentFlags` parameter. This method uses a bitwise OR to set the `REQUEST_ACTIVITY_ENDED_CALLBACK` flag.
- The `setRequestActivityUnreferencedCallback` method.
  This method returns the result of setting the `REQUEST_ACTIVITY_UNREFERENCED_CALLBACK` flag in the supplied `currentFlags` parameter. This method uses a bitwise OR to set the `REQUEST_ACTIVITY_UNREFERENCED_CALLBACK` flag.
- The `toString` method.
  This method returns a string representation of the flags which are set in the supplied `flags` parameter.

15.18 EventFlags class

The EventFlags class defines a set of flags that relate to Events. These flags are used by resource adaptor objects when firing events to the SLEE via the `SleeEndpoint` interface, and also by the SLEE in the optional event processing callback methods invoked on resource adaptor objects. The flags identify SLEE functions that a Resource Adaptor may make use of with respect to an Event that the flags are combined with, or the result of event processing in the SLEE.

The public interface of the EventFlags class is as follows:

```
package javax.slee.resource;

public final class EventFlags {
        // constants
        public static final int NO_FLAGS = …;
        public static final int STANDARD_FLAGS_MASK …;
        public static final int VENDOR_FLAGS_MASK …;

        // flags
        public static final int SLEE_MAY_MARSHAL …;
        public static final int REQUEST_PROCESSING_SUCCESSFUL_CALLBACK …;
        public static final int REQUEST_PROCESSING_FAILED_CALLBACK …;
        public static final int REQUEST_EVENT_UNREFERENCED_CALLBACK …;
        public static final int SBB_PROCESSED_EVENT …;

        // methods to test flags
        public static boolean hasNoFlags(int flags) { … }
        public static boolean hasStandardFlags(int flags) { … }
        public static boolean hasVendorFlags(int flags) { … }
        public static boolean hasFlag(int flag, int flagsToTestFor) { … }
        public static boolean hasSleeMayMarshal(int flags) { … }
        public static boolean hasRequestProcessingSuccessfulCallback(int flags) { … }
        public static boolean hasRequestProcessingFailedCallback(int flags) { … }
        public static boolean hasRequestEventUnreferencedCallback(int flags) { … }
        public static boolean hasSbbProcessedEvent(int flags) { … }

        // methods to set flags
        public static int setSleeMayMarshal(int currentFlags) { … }
        public static int setRequestProcessingSuccessfulCallback(int currentFlags) { … }
        public static int setRequestProcessingFailedCallback(int currentFlags) { … }
        public static int setRequestEventUnreferencedCallback(int currentFlags) { … }
        public static int setSbbProcessedEvent(int currentFlags) { … }

        public static String toString(int flags) { … }
}
```

The following constants are defined:

- `NO_FLAGS`.
  If a flags parameter is set to this value it indicates that there are no flags set.

- `STANDARD_FLAGS_MASK`.
  This constant defines a bit-mask that covers the range for current and future SLEE specification defined flag values. Bits 0-23 are reserved for use for the SLEE specification.
- `VENDOR_FLAGS_MASK`.
  This constant defines a bit-mask that covers the range for vendor extension flag values. Bits 24-31 are reserved for use by SLEE vendors.

The following flags are defined:

- The `SLEE_MAY_MARSHAL` flag.
  If the `SLEE_MAY_MARSHAL` flag is set when an Event is fired to the SLEE then the SLEE implementation may, at its discretion, use the Resource Adaptor's Marshaler object to marshal and unmarshal the Event object. Refer to Section 15.15 for a description of the `Marshaler` interface implemented by the Marshaler object.

- The `REQUEST_PROCESSING_SUCCESSFUL_CALLBACK` flag.
  If the `REQUEST_PROCESSING_SUCCESSFUL_CALLBACK` flag is set when an Event is fired to the SLEE then the SLEE implementation must invoke the `eventProcessingSuccessful` callback method on the resource adaptor object if processing of the Event was successful. If this flag is not set then the SLEE implementation will not invoke the `eventProcessing-Successful` callback method for that Event in the case where event processing is successful. For more information refer to the description of the `eventProcessingSuccessful` callback method in Section 15.12.6.1.
- The `REQUEST_PROCESSING_FAILED_CALLBACK` flag.
  If the `REQUEST_PROCESSING_FAILED_CALLBACK` flag is set when an Event is fired to the SLEE then the SLEE implementation must invoke the `eventProcessingFailed` callback method on the resource adaptor object if processing of the Event was not successful. If this flag is not set then the SLEE implementation will not invoke the `eventProcessingFailed` callback method for that Event in the case where event processing fails. For more information refer to the description of the `eventProcessingFailed` callback method in Section 15.12.6.2.
- The `REQUEST_EVENT_UNREFERENCED_CALLBACK` flag.
  If the `REQUEST_EVENT_UNREFERENCED_CALLBACK` flag is set when an Event is fired to the SLEE then the SLEE implementation must invoke the `eventUnreferenced` callback method on the resource adaptor object when the Event object becomes unreferenced in the SLEE. If this flag is not set then the SLEE implementation will not invoke the `eventUnreferenced` callback method for that Event. For more information refer to the description of the `eventUn-referenced` callback method in see Section 15.12.6.3.
- The `SBB_PROCESSED_EVENT` flag.
  The SLEE sets the `SBB_PROCESSED_EVENT` flag if the SLEE invoked an event handler method on at least one SBB for the Event. A Resource Adaptor should not set this flag, instead it should test if this flag is set in the `eventFlags` parameter of an event processing callback method. The SBB_PROCESSED_EVENT flag will be cleared by a `SleeEndpoint`'s fire-event method if it receives an `eventFlags` parameter with this flag set..

The following methods are defined:

- The `hasNoFlags` method.
  This method returns `true` if the supplied `flags` parameter is equal to the constant `NO_FLAGS`.
- The `hasStandardFlags` method.
  This method returns `true` if the supplied `flags` parameter has one or more flags set which are in the `STANDARD_FLAGS_MASK` range (i.e. one or more of bits 0-23 are set).

- The `hasVendorFlags` method.
  This method returns `true` if the supplied `flags` parameter has one or more flags set which are in the `VENDOR_FLAGS_MASK` range (i.e. one or more of bits 24-31 are set).
- The `hasFlags` method.
  This method returns `true` if all the flags indicated by the `flagsToTestFor` parameter are set in the `flags` parameter.
- The `hasSleeMayMarshal` method.
  This method returns `true` if the supplied `flags` parameter has the `SLEE_MAY_MARSHAL` flag set.
- The `hasRequestProcessingSuccessful` method.
  This method returns `true` if the supplied `flags` parameter has the `REQUEST_PROCESSING_SUCCESSFUL` flag set.
- The `hasRequestProcessingFailed` method.
  This method returns `true` if the supplied `flags` parameter has the `REQUEST_PROCESSING_FAILED` flag set.
- The `hasRequestEventReferenceReleasedCallback` method.
  This method returns `true` if the supplied `flags` parameter has the `REQUEST_EVENT_REFERENCE_RELEASED_CALLBACK` flag set.
- The `hasSbbProcessedEvent` method.
  This method returns `true` if the supplier `flags` parameter has the `SBB_PROCESSED_EVENT` flag set.
- The `setSleeMayMarshal` method.
  This method returns the result of setting the `SLEE_MAY_MARSHAL` flag in the supplied `currentFlags` parameter. This method uses a bitwise OR to set the `SLEE_MAY_MARSHAL` flag.
- The `setRequestProcessingSuccessful` method.
  This method returns the result of setting the `REQUEST_PROCESSING_SUCCESSFUL_CALLBACK` flag in the supplied `currentFlags` parameter. This method uses a bitwise OR to set the `REQUEST_PROCESSING_SUCCESSFUL_CALLBACK` flag.
- The `setRequestProcessingFailed` method.
  This method returns the result of setting the `REQUEST_PROCESSING_FAILED_CALLBACK` flag in the supplied `currentFlags` parameter. This method uses a bitwise OR to set the `REQUEST_PROCESSING_FAILED_CALLBACK` flag.
- The `setRequestEventUnreferencedCallback` method.
  This method returns the result of setting the `REQUEST_EVENT_UNREFERENCED_CALLBACK` flag in the supplied `currentFlags` parameter. This method uses a bitwise OR to set the `REQUEST_EVENT_UNREFERENCED_CALLBACK` flag.
- The `setSbbProcessedEvent` method.
  This method returns the result of setting the `SBB_PROCESSED_EVENT` bit in the supplied `currentFlags` parameter. This method uses a bitwise OR to set the `SBB_PROCESSED_EVENT` flag. Note that this method is not intended to be invoked by Resource Adaptors.
- The `toString` method.
  This method returns a string representation of the flags which are set in the supplied `flags` parameter.

15.19 Resource adaptor entity and Activity Context interactions

The SLEE and Resource adaptor entities collaborate to manage the lifecycle of an Activity object and the Activity Context associated with an Activity object. The SLEE interacts with a resource adaptor entity in the following ways:

- The SLEE can inform the resource adaptor entity responsible for the Activity object when the Activity object's Activity Context is no longer attached to any SBB entities and is no longer referenced by any SLEE Facilities. This enables the resource adaptor entity to implicitly end the Activity object when the Activity object's Activity Context is no longer referenced. See the `ResourceAdaptor` interface's `activityUnreferenced` method described in section 15.12.6.5.
- The resource adaptor entity informs the SLEE when an Activity object has ended. This allows the SLEE to move the Activity object's Activity Context to the Ending state. This is achieved by invoking either the `endActivity` or `endActivityTransacted` method on a `SleeEndpoint` object, as described in sections 15.14.6 and 15.14.7.
- The SLEE informs the resource adaptor entity responsible for the Activity object when the Activity object's Activity Context moves from the Ending state into the Invalid state. This allows the resource adaptor object to release any references to internal state related to the Activity. See the `ResourceAdaptor` interface's `activityEnded` method described in section 15.12.6.4.

15.20 Activity Context Interface Factories

A resource adaptor type specifies the Java interface for an Activity Context Interface Factory. The SLEE is responsible for implementing the Activity Context Interface Factory. The SLEE uses the `getActivityHandle` method on the `ResourceAdaptor` interface in order to implement the `get<<name>>ActivityContextInterface` methods specified by the Activity Context Interface Factory. The following pseudo code illustrates a typical implementation of a `get<<name>>ActivityContextInterface` method.

```
// activity                    is the Activity object passed to the method.
// raEntitiesofRAType          is the set of RA entities existing in the SLEE that have
                               been created from RAs that implement the RA type that
                               defined the Activity Context Interface Factory being
                               implemented.
// raEntity                    is an RA entity
// raEntity.rao                is a resource adaptor object of the RA entity
// handle                      is an Activity Handle for the activity

for each raEntity in raEntitiesOfRAType
        // query the RA entity to see if it owns the activity
        handle = raEntity.rao.getActivityHandle(activity)
        if handle != null
                // the activity is owned by this raEntity
                map handle to Activity Context
                return Activity Context Interface object for Activity Context
        endif
endfor

// activity is unknown
throw javax.slee.UnrecognizedActivityException
```

Refer to Section 15.12.5.3 for details on the `getActivityHandle` method of the `ResourceAdaptor` interface and the responsibility of the Resource Adaptor Developer when implementing this method.

15.21 Accessing Activity Context Interface Factories from within an SBB

An SBB can explicitly lookup an object that implements the Activity Context Interface Factory interface of a resource adaptor type in its JNDI environment. It can use this object to obtain Activity Context Interface objects from the Activity objects created by resource adaptor entities of the resource adaptor type. This

requires a `resource-adaptor-type-binding` element in the SBB's deployment descriptor to bind an Activity Context Interface Factory object into the JNDI component environment of the SBB.

15.22 Accessing Profiles from within a Resource Adaptor

A resource adaptor entity may use profiles to read, write, create, or remove state in a SLEE-managed manner. This state can be arbitrary; for example per-activity state, administrative information which could change at runtime, subscription information, etc. A Resource Adaptor must follow the behavior outlined in the Profile chapter when using profiles (see Chapter 10).

- A resource adaptor uses `ProfileTable` objects and Profile Local interface objects to interact with profiles. `ProfileTable` objects can be obtained from the `ResourceAdaptorContext`. Profile Local interface objects are obtained from a `ProfileTable` object.
- Profiles and profile state must be created, deleted and updated within a transaction. The resource adaptor can use the `SleeTransactionManager` start and commit transactions when necessary (see section 9.14.2).

15.22.1 Profiles for storing per-activity state

A Resource Adaptor may use Profiles to store Activity related state. For example, if an Activity represents a "session" then the start time of the session could be stored in a Profile.

- Per-activity state can be created by a resource adaptor entity by using some unique activity identifier as part of each profile's name.
- The resource adaptor entity has the responsibility of creating any profiles that it needs to use related to an activity.
- The Resource Adaptor has the responsibility of removing any profiles which it uses related to an activity.

15.22.2 A Resource Adaptor CMP Profile Specification example

The Profile CMP interface of an example Resource Adaptor's Foo Profile Specification is as follows:

```
package com.foobar;

public interface FooProfileCMP {
        public String getFooCMP();
        public void setFooCMP(String value);
}
```

The Profile Local interface of this Profile Specification is as follows:

```
package com.foobar;

import javax.slee.profile.ProfileLocalObject;

public interface FooProfileLocal extends ProfileLocalObject, FooProfileCMP {
}
```

A Profile Local interface is defined to enable a read and write view to the resource adaptor for the CMP state attributes.

The Profile Management interface of this Profile Specification is as follows:

```
package com.foobar;

public interface FooProfileManagement extends FooProfileCMP {
}
```

A Profile Management interface is defined so that the Administrator may remove any per-activity state which the Resource Adaptor inadvertently does not remove.

A Profile abstract class are not defined as additional business or management methods are not defined for this Profile. The SLEE implements the CMP fields of the Profile CMP interface.

This profile specification does not define any static query methods, therefore does not define a Profile Table interface extending the default `ProfileTable` interface defined by the SLEE specification.

The `profile-spec` element that describes this Profile Specification is as follows:

```
<profile-spec>
        …
        <profile-spec-name> FooProfileSpec </profile-spec-name>
        <profile-spec-vendor> com.foobar </profile-spec-vendor>
        <profile-spec-version> 3.1.1 </profile-spec-version>
        <profile-classes>
                <profile-cmp-interface >
                        <description> … </description>
                        <profile-cmp-interface-name>
                                com.foobar.FooProfileCMP
                        </profile-cmp-interface-name>
                </profile-cmp-interface>
                <profile-local-interface>
                        <description> … </description>
                        <profile-local-interface-name>
                                com.foobar.FooProfileLocal
                        </profile-local-interface-name>
                </profile-local-interface>
                <profile-management-interface>
                        <description> … </description>
                        <profile-management-interface-name>
                                com.foobar.FooProfileManagement
                        </profile-management-interface-name>
                </profile-management-interface>
        </profile-classes>
        …
</profile-spec>
```

15.23 Accessing resource objects from within an SBB

In this section, a resource object refers to an object created by a resource adaptor entity. An SBB can obtain access to a resource object in the following ways.

- Reference to Event objects passed to an SBB's event handler method.
- Reference to Activity objects via the `ActivityContextInterface` parameter passed in an SBB's event handler method.
- Receiving an Activity object (owned by a resource adaptor entity) embedded in an Event object.
  An SBB object of the SBB may receive an Event fired by a resource adaptor entity. The event object may contain direct or indirect references to Activity objects owned by the resource adaptor entity. The SBB object can interact with the resource adaptor entity through one or more of these Activity objects. This does not require a `resource-adaptor-entity-binding` element in the SBB's deployment descriptor. An SBB does not have to reference a resource adaptor entity or resource adaptor type of the resource adaptor entity in its deployment descriptor to receive events fired by the resource adaptor entity. For example, root SBB entities of this SBB will receive initial events from the resource adaptor entity if the resource adaptor entity fires events of the initial event types of the SBB and these Events are fired on addresses stored in the Address Profile Table of the Service that specifies this SBB as its root SBB.
- Looking up a resource object in the JNDI component environment of the SBB.
  The SBB can explicitly lookup an object that implements the resource adaptor interface of the resource adaptor type in its JNDI component environment. It can use this object to interact with the resource adaptor entity. This requires a `resource-adaptor-entity-binding` element in the SBB's deployment descriptor to establish the resource adaptor entity binding in the JNDI component environment of the SBB.

- Receiving input parameters passed through its SBB local interface.

15.23.1 Accessing multiple resource adaptor entities

An SBB may use multiple resource adaptor entities of the same resource adaptor type or different resource adaptor types. The SBB Developer should specify the JNDI names of the resource adaptor interface objects (through the `resource-adaptor-object-name` element) and resource adaptor type defined Activity Context Interface Factory objects (through the `activity-context-interface-factory-name` element) accessed by the SBB. In addition, the SBB Developer should also specify the requirements or properties of the objects that should be bound to the specified JNDI names in the `description` element of the corresponding `resource-adaptor-entity-binding` and `resource-adaptor-type-binding` elements.

The Service Deployer uses these descriptions to bind the appropriate resource adaptor entities and resource adaptor types to the specified JNDI names.

The SBB looks up the specified JNDI names for the resource adaptor objects and resource adaptor type defined Activity Context Interface Factory objects that it requires.

15.23.2 Example code

This following example shows how an SBB may obtain a reference to two resource adaptor objects belonging to two different resource adaptor entities of the same resource adaptor type, and obtain the SBB Activity Context Interface object for one of the Activity objects of this resource adaptor type.

The SBB deployment descriptor for this SBB is as follows:

```
<sbb>
        …
        <resource-adaptor-type-binding>
                <description>
                        This SBB expects the JCC 1.1 resource type to be
                        bound to the JNDI name specified below.
                </description>
                <resource-adaptor-type-ref>
                        <resource-adaptor-type-name>
                                JCC
                        </resource-adaptor-type-name>
                        <resource-adaptor-type-vendor>
                                javax.csapi.cc.jcc
                        </resource-adaptor-type-vendor>
                        <resource-adaptor-type-version>
                                1.1
                        </resource-adaptor-type-version>
                </resource-adaptor-type-ref>
                <activity-context-interface-factory-name>
                        slee/resource/jcc/1.1/activitycontextinterfacefactory
                </activity-context-interface-factory-name>

                <resource-adaptor-entity-binding>
                        <description>
                                This SBB expects a JCC 1.1 resource that has the
                                following properties to be bound to the JNDI name
                                specified below.
                                1. …
                                2. …
                        </description>
                        <resource-adaptor-object-name>
                                slee/resources/jcc/1.1/jcc_1
                        </resource-adaptor-object-name>
                        <resource-adaptor-entity-link>
                                FooBarJCC
                        </resource-adaptor-entity-link>
                </resource-adaptor-entity-binding>
                …
                <resource-adaptor-entity-binding>
```

```
                        <description>
                                This SBB expects a JCC 1.1 resource that has the
                                following properties to be bound to the JNDI name
                                specified below.
                                1. …
                                2. …
                                3. …
                                4. …
                        </description>
                        <resource-adaptor-object-name>
                                slee/resources/jcc/1.1/jcc_2
                        </resource-adaptor-object-name>
                        <resource-adaptor-entity-link>
                                CuckooJCC
                        </resource-adaptor-entity-link>
                </resource-adaptor-entity-binding>
        </resource-adaptor-type-binding>
        …
</sbb>
```

The SBB code that accesses the resource adaptor objects, the Activity Context Interface Factory object of the resource adaptor type of these resource adaptor entities, and obtains an SBB Activity Context Interface object from a generic Activity Context Interface object is as follows.

```
        …
        // obtain the SBB component environment naming context.
        Context initCtx = new InitialContext();
        Context myEnv = (Context)initCtx.lookup("java:comp/env");
        …
        // get first JCC resource adaptor object
        JccProvider jcc1 = (JccProvider) myEnv.lookup("slee/resources/jcc/1.1/jcc_1");
        …
        // get second JCC resource adaptor object
        JccProvider jcc2 = (JccProvider) myEnv.lookup("slee/resources/jcc/1.1/jcc_2");
        …
        // create a call on jcc1
        JccCall call = jcc1.createCall();
        …
        // get ActivityContectInterfaceFactory object for JCC
        JccActivityContextInterfaceFactory jccACIFactory =
                (JccActivityContextInterfaceFactory)
                myEnv.lookup("slee/resources/jcc/1.1/activitycontextinterfacefactory");
        …
        // get generic Activity Context Interface object for call
        ActivityContextInterface ac = jccACIFactory.getActivityContext(call);
        …
        // get SBB Activity Context Interface object from
        // generic Activity Context Interface object
        FooSbbActivityContextInterface fooAC = asSbbActivityContextInterface(ac);
        …
```

Chapter 16 Libraries

Added in 1.1: Libraries are a new feature of JAIN SLEE 1.1.

A library is a SLEE component that provides some common functionality to other components installed in the SLEE. A library can depend on other libraries in the SLEE, but cannot depend on other types of components such as SBBs and Resource Adaptors. This means that a library is typically a "passive" component used by other components to support their functionality, rather than an "active" component which initiates interaction with other components in the SLEE.

A library consists of a collection of Java classes. A library may also include static resources loaded at runtime, such as XML documents, resource bundles, and so on. These classes and resources may be included directly in a library's component jar file, or may be included in pre-packaged jars which are in turn included in the library component jar and referenced in the library's deployment descriptor. The classes and resources available to an individual library is the sum of:

- Any classes and resources contained directly within the library component jar.
- The classes and resources contained within the constituent jars of the library component jar, and referenced by the library's deployment descriptor.

Together, all of these classes and resources are loaded by the same classloader within a single JVM.

Rationale for the addition of Library components to the SLEE model:

Library components supersede the use of manifest classpath entries to reference jars containing common functionality. The use of manifest classpaths in the SLEE to reference dependent jars was ill-defined and also very limited in scope. Use of manifest classpath jars between components within the same deployable unit was undefined, and as manifest classpath jars had no component identity, they could not be shared between deployable units. Library components solve these problems by defining a formal description for what constitutes a library, and providing libraries with component identity that may be referenced by all other components in the SLEE, including other libraries.

16.1 Library identifiers

A library is uniquely identified by the name, vendor, and version of the library. Two libraries with the same name, vendor, and version are considered to be the same library. Different libraries are assumed to be incompatible, meaning that one cannot be substituted for the other.

16.2 Library deployment descriptor

A library deployment descriptor contains a `library` element. This element contains the following sub-elements:

- A `description` element.
  This is an optional informational element.
- A `library-name` element, a `library-vendor` element, and a `library-version` element.
  These elements uniquely identify the library.
- Zero or more `library-ref` elements.
  Each library component required in turn by this library must be identified by a `library-ref` element. Each `library-ref` element has the following sub-elements:
    - A `description` element.
      This is an optional informational element.

  - A `library-name` element, a `library-vendor` element, and a `library-version` element.
    These elements uniquely identify the dependent library.

- Zero or more `jar` elements.
  Each constituent jar that contains classes or resources to be included in the library must be identified by a `jar` element. Each `jar` element has the following sub-elements:
  - A `description` element.
    This is an optional informational element.
  - A `jar-name` element.
    Each `jar` element contains the name of a constituent jar file included in the library component jar file. This name is relative to the root of the library component jar file.
  - A `security-permissions` element.
    This element is optional and contains sub-elements that identify additional security permissions that should be granted to the classes in the constituent jar file.
    - A `description` element.
      This is an optional informational element.
    - A `security-permission-spec` element.
      This element identifies the security permission policies used by the jar file classes. The syntax of this element is defined at the following URL: `http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html#FileSyntax`.
      Note: If the `codeBase` argument is not specified for a `grant` entry in the provided security permission specification, the SLEE assumes the code base to be the constituent jar file, and the security permissions specified in the `grant` entry are granted to all classes loaded from the constituent jar file. If the `codeBase` argument is specified for a `grant` entry, the specified path is taken to be relative to the root directory of the constituent jar within the library component jar, but its use is otherwise undefined by the SLEE specification.

16.3 Library jar file

The library jar file is the standard format for the packaging of one or more libraries. Libraries packaged in this format and included in a deployable unit may be deployed directly into a compliant SLEE. It must include the following:

- A library jar deployment descriptor.
  - The library jar deployment descriptor is stored with the name `META-INF/library-jar.xml` in the library jar file.
  - The root element of the library jar deployment descriptor is a `library-jar` element. This element contains the following sub-elements.
    - A `description` element.
      This is an optional informational element.
    - One or more `library` elements.
      Each of these elements is a library deployment descriptor as defined in Section 16.2.
    - A `security-permissions` element.
      This element is optional and contains sub-elements that identify security permissions of the library classes included directly in the library component jar file.
      - A `description` element.
        This is an optional informational element.

- A `security-permission-spec` element.
  This element identifies the security permission policies used by the library classes. The syntax of this element is defined at the following URL:
  `http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html#FileSyntax`.
  Note: If the `codeBase` argument is not specified for a `grant` entry in the provided security permission specification, the SLEE assumes the code base to be the library component jar file, and the security permissions specified in the `grant` entry are granted to all classes loaded from the library component jar file, i.e. they apply to all library components defined in the library component jar file. The security permissions are not granted to classes contained in constituent jar files or from any other dependent component jar required by the libraries defined in the library deployment descriptor. If the `codeBase` argument is specified for a `grant` entry, the specified path is taken to be relative to the root directory of the library component jar within the deployable unit jar, but its use is otherwise undefined by the SLEE specification.

- Class files of the libraries specified by the `library` elements of the `library-jar` element.
  - The library jar file must contain directly the class or resource files required for the libraries where those classes and resources are not contained in any referenced jar.
  - The library jar file must include any constituent jar referenced by `jar-name` elements in the library component's deployment descriptor.
  - The library jar file must contain, either by inclusion or by reference, the class files for all the classes and interfaces that the Java types of the library depend on, except J2SE classes and SLEE classes, and classes contained in the library component jar files of the libraries referenced by this library component jar. This includes their superclasses and superinterfaces, dependent classes, and the classes and interfaces used as method parameters, results, and exceptions.

16.3.1 Library jar file example

The following example illustrates a library jar file that declares a library.

The deployment descriptor for the example library jar file is as follows:

```
<library-jar>
        …
        <library>
                <description> … </description>
                <library-name>
                        JCC
                </library-name>
                <library-vendor>
                        javax.csapi.cc.jcc
                </library-vendor>
                <library-version>
                        1.1
                </library-version>
                <jar>
                        <description> … </description>
                        <jar-name>
                                jars/jcc-1.1.jar
                        </jar-name>
                </jar>
        </library>
</library>
```

The content of the library jar file is as follows:

```
META-INF/library-jar.xml
META-INF/MANIFEST.MF
…
jars/jcc-1.1.jar
…
```

Appendix A Event Router Formal Model

Changed in 1.1: This appendix has been updated to include modelling of firing events to a particular Service ID.

A.1 Notation

Z is the set of integers. **R** is the set of real numbers.

P(*s*), where *s* is a set, is called the *power set* of s. It is defined as the set of all subsets of *s*

$$P(s) = \{x : x \subseteq s\}.$$

x_i refers to the i^{th} component of the tuple **x**. If a component of a tuple **x** is itself a tuple, the notation $(x_i)_j$ indicates the j^{th} component of the i^{th} component of **x**. For example, if

$$\mathbf{x} = ((a,b)\ ,(c,d))$$

then

$$(x_1)_1 = a,$$
$$(x_1)_2 = b,$$
$$(x_2)_1 = c,$$
$$(x_2)_2 = d,$$

The *minarg* function is defined as follows:

$$\min\arg_{x \in S} f(x) = y \in S \text{ such that } f(y) = \min_{x \in S} f(x)$$

A.2 Definitions

A.2.1 Events

Let *EventType* be the set of event types and let *Event* be the set of events. An *event* **e** is an ordered 5-tuple,

$$(e_1,e_2,e_3,e_4,e_5) \in \mathbf{Z} \times EventType \times ActivityContext \times Address \times P(Service),$$

where e_1 is the *identity* of the event, e_2 is the *type* of event, e_3 is the *Activity Context*[46] on which the event was fired, e_4 is the SLEE-determined *address* for the event, and e_5 is a set of services.

A.2.2 Child relations

Let *ChildRelation* be the set of child relations. A *child relation* **cr** is an ordered pair,

$$(cr_1, cr_2) \in \mathbf{Z} \times SBBComponent\ ,$$

where cr_1 is the *default priority* of the child relation and cr_2 is the *child SBBcomponent*[47] of the child relation.

A.2.3 SBB Components

Let *SBBComponent* be the set of all SBB components. An *SBB component* **c** is a 5-tuple

[46] *ActivityContext* is defined in Section A.2.7.
[47] *SBBComponent* is defined in Section A.2.3.

$$(c_1, c_2, c_3, c_4, c_5) \in P(ChildRelation) \times P(EventType) \times P(EventType) \times P(EventType) \times P(EventName)$$

where c_1 is the set of *child relations* of the SBB component, c_2 is the set of *initial event types* of the SBB component, c_3 is the set of *received event types* of the SBB component, c_4 is the set of *mask-on-attach event types* of the SBB component, and c_5 is the set of *mask-on-attach event names* of the SBB component. For a given SBB component **c**,

$$c_2, c_4 \subseteq c_3 .$$

A.2.4 Child relation objects

Let *ChildRelationObject* be the set of all child relation objects.

Let *ExistingChildObject* (*t*) be the set of child relation objects that are said to 'exist' at time *t*. A *child relation object* **cro** that exists at time *t* is an ordered pair,

$$(cro_1, cro_2) \in ChildRelation \times P(ExistingSBB(t)) ,$$

where cro_1 is the *child relation* of the child relation object and cro_2 is the set of *child SBB entities*[48] of the child relation object.

A.2.5 SBB Entities

Let *SBBEntity* be the set of all SBB entities. An *SBB entity* **x** is an ordered triple,

$$(x_1, x_2, x_3, x_4) \in \mathbf{Z} \times SBBComponent \times P(Name) \times Service,$$

where x_1 is the *identity* of the SBB entity, x_2 is the *type* of the SBB entity, x_3 is a set containing the *convergence name*[49] of the SBB entity, and x_4 is the Service that the SBB entity is a part of.

We define

$$ExistingSBB(t) : \mathbf{R} \quad \mapsto \quad P(SBBEntity)$$

as a relation which defines the set of SBB entities that are said to 'exist' at time *t*. For each SBB entity $\mathbf{x} \in ExistingSBB(t)$, there are relations

$$\begin{aligned}
priority_{\mathbf{x}}(t) &: \mathbf{R} \mapsto \mathbf{Z} \\
children_{\mathbf{x}}(t) &: \mathbf{R} \mapsto P(ExistingSBB(t)) \\
child_objects_{\mathbf{x}}(t) &: \mathbf{R} \mapsto P(ChildRelationObjects(t)) \\
parent_{\mathbf{x}}(t) &: \mathbf{R} \mapsto P(ExistingSBB(t)) \\
activities_{\mathbf{x}}(t) &: \mathbf{R} \mapsto P(ExistingContext(t)) \\
attachment_count_{\mathbf{x}}(t) &: \mathbf{R} \mapsto \mathbf{Z} \\
event_type_{\mathbf{x}}(name, t) &: EventName \times \mathbf{R} \mapsto EventType
\end{aligned}$$

describing the *priority*, *children*, *child relation objects*, *parent*, *Activity Contexts*[50], and *event type mapping* for that SBB entity at time *t*.

The image of $parent_{\mathbf{x}}(t)$ is the set $\{x \in P(ExistingSBB(t)) : |x| \leq 1\}$, i.e. the elements of $P(ExistingSBB(t))$ that are sets containing at most one element, and an SBB entity **x** that is a root SBB

---

[48] *SBB entities* are discussed in Section A.2.5.
[49] *Convergence names* are discussed in Section A.3.2.
[50] *ExistingContext*(*t*) is defined in Section A.2.7.

entity for a Service will have $parent_{\mathbf{x}}(t) = \{\}$. (For a time *t'* such that $\mathbf{x} \notin ExistingSBB(t')$, the above relations are undefined.) We let

$$SleeSBB = \bigcup_{t \in \mathbf{R}} ExistingSBB(t)$$

be the set of all SBB entities that exist during the lifetime of the SLEE.

A.2.6 Services

Let *Service* be the set of all Services. A *Service* **v** is a 4-tuple,

$$(v_1, v_2, v_3, v_4) \in \mathbf{Z} \times \mathbf{Z} \times SBBComponent \times ChildRelationObject ,$$

where v_1 is the *identity* of the Service, v_2 is the *default priority* of the Service, v_3 is the *root SBB component* for the Service, and v_4 is the *child relation object* for the Service. *SleeService*(*t*) is a relation which specifies the set of Services that are known to the SLEE at time *t*:

$$SleeService(t) : \mathbf{R} \quad \mapsto \quad P(Service)$$

A.2.7 Activity Contexts

Let *ActivityContext* be the set of all Activity Contexts. Let *ExistingContext*(*t*) be the set of Activity Contexts that are said to 'exist' at time *t*. An *Activity Context* **ac** is an ordered pair,

$$(ac_1, ac_2) \in \mathbf{Z} \times Activity ,$$

where ac_1 is the *identity* of the Activity Context and ac_2 is the *activity* of the Activity Context, together with relations

$$\begin{aligned}
attached_{\mathbf{ac}}(t) &: \mathbf{R} \quad \mapsto \quad P(ExistingSBB(t)) \\
delivered_{\mathbf{ac}}(\mathbf{e}, t) &: Event \times \mathbf{R} \quad \mapsto \quad P(ExistingSBB(t)) \\
type_mask_{\mathbf{ac}}(\mathbf{sbbe}, t) &: ExistingSBB(t) \times \mathbf{R} \quad \mapsto \quad P(EventType) \\
name_mask_{\mathbf{ac}}(\mathbf{sbbe}, t) &: ExistingSBB(t) \times \mathbf{R} \quad \mapsto \quad P(EventName)
\end{aligned}$$

where $attached_{\mathbf{ac}}(t)$ is the set of SBB entities that are said to be 'attached' to the Activity Context at time *t* and $delivered_{\mathbf{ac}}(\mathbf{e}, t)$ is the set of SBB entities to which the event **e** has been delivered at time *t*.

A.3 Further Properties

A.3.1 Uniqueness of SBB Entities

The identity of an SBB entity is unique:

$$\forall \mathbf{x}, \mathbf{y} \in SleeSBB, \quad x_1 = y_1 \quad \Leftrightarrow \quad \mathbf{x} = \mathbf{y}$$

A.3.2 Convergence names

Only root SBB entities have a convergence name:

$$\forall t \in \mathbf{R}, \quad \mathbf{s} \in RootSBB(t) \Leftrightarrow |s_3| = 1$$
$$\forall t \in \mathbf{R}, \quad \mathbf{s} \in ExistingSBB(t) - RootSBB(t) \Leftrightarrow |s_3| = \{\}$$

A.3.3 Delivered and attached

$$\forall t \in \mathbf{R}, \mathbf{e} \in Event, \quad delivered_{\mathbf{ac}}(\mathbf{e}, t) \subseteq attached_{\mathbf{ac}}(\mathbf{e}, t)$$

A.3.4 Descendents and ancestors

An SBB entity $\mathbf{d} \in ExistingSBB(t)$ is a *descendent* of $\mathbf{a} \in ExistingSBB(t)$ at time *t* if and only if $\exists \mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_N \in ExistingSBB(t)$ such that

$$parent_{\mathbf{s}_1}(t) = \{\mathbf{s}_2\}, \quad parent_{\mathbf{s}_2}(t) = \{\mathbf{s}_3\}, \quad ..., \quad parent_{\mathbf{s}_{N-1}}(t) = \{\mathbf{s}_N\}$$

where $\mathbf{d} = \mathbf{s}_1$ and $\mathbf{a} = \mathbf{s}_N$. If **d** is a descendent of **a**, then **a** is an *ancestor* of **d**. The set of all descendents of an SBB entity **x** at time *t* is written

$$descendents_{\mathbf{x}}(t) = \{\mathbf{y} : \mathbf{y} \text{ is a descendent of } \mathbf{x} \text{ at time } t\}$$

A.3.5 Priority of attached SBB entities with respect to an Activity Context and event

Let

$$deliverable_{\mathbf{ac}}(\mathbf{e},t) : Event \times \mathbf{R} \quad \mapsto P(ExistingSBB(t))$$

be the set of SBB entities attached to the Activity Context **ac** at time *t* that have the event type of **e** in the SBB component's received event set and have not received or masked the event **e** and the SBB entities Service is an element of the events service set, defined as:

$$deliverable_{\mathbf{ac}}(\mathbf{e},t) = attached_{\mathbf{ac}}(t) - \{\{\mathbf{x} : e_2 \in type\_mask_{\mathbf{ac}}(\mathbf{x},t) \quad \vee \quad e_2 \notin (x_2)_3\} \quad \cup \quad delivered_{\mathbf{ac}}(\mathbf{e},t)\} \cap \quad \{x : x_4 \in e_5\}$$

Let

$$origin(\mathbf{x},t) : SBBEntity \times \mathbf{R} \quad \mapsto \quad P(ExistingSBB(t))$$

be the set consisting of the (unique) ancestor of **x** which has no parent, defined as:

$$origin(\mathbf{x},t) = \begin{cases} \{\mathbf{x}\} \text{ if } parent_{\mathbf{x}}(t) = \{\}, \quad \text{otherwise} \\ \{\mathbf{sbbe} : \mathbf{x} \in descendents_{\mathbf{sbbe}}(t) \wedge parent_{\mathbf{sbbe}}(t) = \{\}\} \end{cases}$$

Let $next\_sbbe_{\mathbf{ac}}(\mathbf{e},t)$ be the set consisting of the SBB entity attached to the Activity Context **ac** at time *t* that will be next to receive the event **e**.

$next\_sbbe_{\mathbf{ac}}(\mathbf{e},t)$ contains at most one SBB entity,

$$\left|next\_sbbe_{\mathbf{ac}}(\mathbf{e},t)\right| \le 1 .$$

Definition of $next\_sbbe_{\mathbf{ac}}(\mathbf{e},t)$ **:** If there are no deliverable SBB entities for the event on the Activity Context, then $next\_sbbe_{\mathbf{ac}}(\mathbf{e},t)$ is the empty set:

$$deliverable_{\mathbf{ac}}(\mathbf{e},t) = \{\} \Rightarrow next\_sbbe_{\mathbf{ac}}(\mathbf{e},t) = \{\} .$$

Otherwise, let *stack* be a sequence of SBB entities.

While

$$origin(deliverable_{\mathbf{ac}}(\mathbf{e},t),t) - stack \neq \{\} ,$$

add

$$\mathbf{y} = \underset{\mathbf{x} \in origin(deliverable_{\mathbf{ac}}(\mathbf{e},t),t) - stack}{\min \arg} priority_{\mathbf{x}}(t)$$

to the end of *stack*.

While $stack \neq \{\}$ and $next\_sbbe_{\mathbf{ac}}(\mathbf{e},t)$ has not been determined:

- Remove the last element, **z**, from *stack*;
- If $\mathbf{z} \in deliverable_{\mathbf{ac}}(\mathbf{e},t)$ then $next\_sbbe_{\mathbf{ac}}(\mathbf{e},t) = \{\mathbf{z}\}$;
- Otherwise, while

  $$children_{\mathbf{z}}(t) - stack \neq \{\},$$

  add

  $$\mathbf{y} = \underset{\mathbf{z} \in children_{\mathbf{z}}(t) - stack}{\min \arg} priority_{\mathbf{z}}(t)$$

  to end of *stack*.

A.3.6 Attachment count

The *attachment count* at time *t* for an SBB entity **x** is defined as

$$attachment\_count_{\mathbf{x}}(t) = \left|activities_{\mathbf{x}}(t)\right| + \sum_{\mathbf{y} \in descendents_{\mathbf{x}}(t)} \left|activities_{\mathbf{y}}(t)\right|$$

A.3.7 Consistency of child-parent relations

All SBB entities that are in $children_{\mathbf{y}}(t)$ must have **y** as their parent, and vice-versa:

$$\forall t \in \mathbf{R}, \mathbf{x}, \mathbf{y} \in ExistingSBB(t), \qquad parent_{\mathbf{x}}(t) = \{\mathbf{y}\} \Leftrightarrow \mathbf{x} \in children_{\mathbf{y}}(t)$$

A.3.8 Uniqueness of SBB entities with respect to child relation objects

An SBB entity can only be in one child relation object:

$$\forall t \in \mathbf{R}, \mathbf{x}, \mathbf{y} \in ExistingChildObject(t), \quad x_2 \cap y_2 = \{\}$$

A.3.9 Inclusion of child SBB entities in child relation objects

All SBB entities that are a child of an SBB entity **sbbe** must belong to a child relation object for **sbbe**:

$$\forall t \in \mathbf{R}, \quad children_{\mathbf{sbbe}}(t) = \bigcup_{\mathbf{cro} \in child\_objects_{\mathbf{sbbe}}(t)} cro_2$$

A.3.10 Root SBB entities

Let $RootSBB(t) : \mathbf{R} \mapsto P(ExistingSBB(t))$ be the set of 'root' SBB entities at time *t*.

$$RootSBB(t) \subseteq ExistingSBB(t)$$

RootSBB(*t*) can also be characterized as

$$RootSBB(t) = origin(ExistingSBB(t), t).$$

A.3.11 Reachability of existing SBB entities from a root SBB entity

SBB entities that exist at a given point in time must be root SBB entities or have an ancestor that is a root SBB entity:

$$ExistingSBB(t) = RootSBB(t) \cup \bigcup_{\mathbf{x} \in RootSBB(t)} descendents_{\mathbf{x}}(t)$$

A.4 Event queue

The SLEE event queue *Q* is modeled as a sequence of events

$$Q = \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3, ...\},\ \mathbf{e}_i \in Event$$

together with the relation

$$event\_time(\mathbf{e}) : Event \quad \mapsto \quad \mathbf{R}$$

that associates each event in the queue with a time such that

$$\forall\, \mathbf{e}_i, \mathbf{e}_j \in Q, \quad i < j \Leftrightarrow event\_time(\mathbf{e}_i) < event\_time(\mathbf{e}_j).$$

A.5 Event routing

This section describes the effect of various SLEE operations on the sets of SBB entities and Activity Contexts. For the sake of argument we assume that the preconditions are evaluated at a time $t_1 < t_{op}$ and the post-conditions at a time $t_2 > t_{op}$ that are sufficiently close to the time t_{op} at which the operation occurs that the operation being described is the only significant event to occur in the interval $[t_1, t_2]$.

The assumed event routing logic is:

```
forever do
    receive event on an Activity Context
    while there are eligible SBB entities do
        deliver event to next eligible SBB entity
    endwhile

    if event is end event for the Activity Context
        terminate Activity Context
    endif
endfor
```

A.5.1 Attach an SBB entity (sbbe) to an Activity Context (ac)

Pre:	none
Post:	$\mathbf{sbbe} \in attached_{\mathbf{ac}}(t_2)$ $\mathbf{ac} \in activities_{\mathbf{sbbe}}(t_2)$ $type\_mask_{\mathbf{ac}}(\mathbf{sbbe}, t_2) = (sbbe_2)_4$ $name\_mask_{\mathbf{ac}}(\mathbf{sbbe}, t_2) = (sbbe_2)_5$ $\mathbf{sbbe} \notin delivered_{\mathbf{ac}}(t_2)$
Description:	If the SBB entity is already attached to the Activity Context, no action is taken. Otherwise, the SBB entity is attached to the Activity Context, the Activity Context is added to the SBB entity's set of Activity Contexts, the event type mask for the SBB entity with respect to the Activity Context is set to the corresponding SBB component's mask-on-attach event types and the event name mask for the SBB entity with respect to the Activity Context is set to the corresponding SBB component's mask-on-attach event names. If the SBB entity is in the Activity Context's delivered set, it is removed from this set.

A.5.2 Detach an SBB entity (sbbe) from an Activity Context (ac)

Pre:	none
Pre:	$\mathbf{sbbe} \notin attached_{\mathbf{ac}}(t_2)$ $\mathbf{ac} \notin activities_{\mathbf{sbbe}}(t_2)$ $type\_mask_{\mathbf{ac}}(\mathbf{sbbe}, t_2) = \{\}$ $name\_mask_{\mathbf{ac}}(\mathbf{sbbe}, t_2) = \{\}$
Description:	If the SBB entity is not attached to the Activity Context, no action is taken. Otherwise, the SBB entity is removed from the Activity Context, the Activity Context is removed

from the SBB entity's set of Activity Contexts, the event type mask for the SBB entity with respect to the Activity Context is cleared and the event name mask for the SBB entity with respect to the Activity Context is cleared.

A.5.3 Mask the events ($names \subseteq EventNames$) for an SBB entity (sbbe) on n Activity Context (ac)

Pre: none

Post:

$type\_mask_{\mathbf{ac}}(\mathbf{sbbe}, t_2) = event\_type_{\mathbf{sbbe}}(names)$

$name\_mask_{\mathbf{ac}}(\mathbf{sbbe}, t_2) = names$

Description: The event name mask for the SBB entity with respect to the Activity Context is set to given event names, and the event type mask for the SBB entity with respect to the Activity Context is set to the corresponding event types.

A.5.4 Remove an SBB entity (sbbe) from a parent SBB entity (p)

Pre: $parent_{\mathbf{sbbe}}(t_1) = \{\mathbf{p}\}$

Post:

$\mathbf{sbbe} \notin ExistingSBB(t_2)$

$\mathbf{ac} \in activities_{\mathbf{sbbe}}(t_1) \Rightarrow \mathbf{sbbe} \notin attached_{\mathbf{ac}}(t_2)$

$\mathbf{s} \in descendents_{\mathbf{sbbe}}(t_1)$

$\quad \Rightarrow \mathbf{s} \notin ExistingSBB(t_2)$

$\forall\, \mathbf{s} \in descendents_{\mathbf{sbbe}}(t_1),$

$\quad \mathbf{ac} \in activities_{\mathbf{s}}(t_1) \Rightarrow \mathbf{s} \notin attached_{\mathbf{ac}}(t_2)$

Description: The SBB entity is removed from the set of existing SBB entities, and the SBB entity is detached from each Activity Context in the SBB entity's set of Activity Contexts. Each descendent SBB entity is removed from the set of existing SBB entities, and is detached from each Activity Context in its set of Activity Contexts.

A.5.5 Create a root SBB entity (sbbe) based on a child relation object (cro) belonging to a Service (s)

Pre: *name* is the convergence name for the SBB entity computed by the SLEE

Post:

$\mathbf{sbbe} \in ExistingSBB(t_2)$

$\mathbf{sbbe} \in cro_2$

$sbbe_2 = (cro_1)_2$

$sbbe_3 = \{name\}$

$sbbe_4 = s$

$parent_{\mathbf{sbbe}}(t_2) = \{\}$

$priority_{\mathbf{sbbe}}(t_2) = (cro_1)_1$

$child\_objects_{\mathbf{sbbe}}(t_2) = \{\}$

$activities_{\mathbf{sbbe}}(t_2) = \{\}$

Description: The SBB entity is added to the set of existing SBB entities and the set of child SBB entities for the child relation object. The SBB component for the SBB entity is the SBB component of the child relation for the child relation object. The SBB entity's parent re-

lation maps to the empty set. The priority of the SBB entity is the priority of the child relation for the child relation object.

A.5.6 Create a non-root SBB entity (sbbe) based on a child relation object (cro) belonging to an SBB entity (p)

Pre: *none*

Post:

$\mathbf{sbbe} \in ExistingSBB(t_2)$

$\mathbf{sbbe} \in cro_2$

$sbbe_2 = (cro_1)_2$

$sbbe_3 = \{\}$

$sbbe_4 = \mathbf{p}_4$

$parent_{\mathbf{sbbe}}(t_2) = \{\mathbf{p}\}$

$priority_{\mathbf{sbbe}}(t_2) = (cro_1)_1$

$child\_objects_{\mathbf{sbbe}}(t_2) = \{\}$

$activities_{\mathbf{sbbe}}(t_2) = \{\}$

Description: The SBB entity is added to the set of existing SBB entities and the set of child SBB entities for the child relation object. The SBB component for the SBB entity is the SBB component of the child relation for the child relation object. The SBB entity's parent relation maps to the SBB entity that the child relation object belongs to. The priority of the SBB entity is the priority of the child relation for the child relation object.

A.5.7 Receipt of event (e) on an Activity Context (ac)

Pre: *names* is a set of convergence names computed by the SLEE.

Post:

$delivered_{\mathbf{ac}}(t_2) = \{\}$

$\forall \mathbf{svc} \in SleeServices(t_1) : e_2 \in (svc_3)_2$

$\exists \mathbf{s} \in ExistingSBB(t_2) :$

$s_3 \in names(\mathbf{e}, \mathbf{ac})$

$e_3 \in activities_{\mathbf{s}}(t_2)$

$\mathbf{s} \in attached_{e_3}(t_2)$

$\mathbf{s}_4 \in e_5$

$\mathbf{s} \notin ExistingSBB(t_1) \Rightarrow$

$\mathbf{s} \in (svc_4)_2$

$s_2 = ((svc_4)_1)_2$

$parent_{\mathbf{s}}(t_2) = \{\}$

$priority_{\mathbf{s}}(t_2) = ((svc_4)_1)_1$

$child\_objects_{\mathbf{s}}(t_2) = \{\}$

$activities_{\mathbf{s}}(t_2) = \{\mathbf{ac}\}$

Description: After receipt of the event, each Service for which the event is an initial event has a root SBB entity instantiated. If the SBB entity was created in response to receipt of the event, its attributes are set to the default values.

A.5.8 Delivery of event (e) on an Activity Context (ac)

Pre: $delivered_{\mathbf{ac}}(\mathbf{e}, t_1) \neq attached_{\mathbf{ac}}(t_1)$

Post:
$next\_sbbe_{\mathbf{ac}}(\mathbf{e}, t_1) \in attached_{\mathbf{ac}}(t_2) \; \forall \, t \in [t_1, t_2]$
$\Leftrightarrow \quad next\_sbbe_{\mathbf{ac}}(\mathbf{e}, t_1) \in delivered_{\mathbf{ac}}(\mathbf{e}, t_2)$

Description: The event is delivered to the SBB entity with the highest priority (as defined in Section A.3.5). After delivery, the SBB entity (if it remained attached to the Activity Context, i.e. did not detach and re-attach) is in the delivered set for the Activity Context with respect to the event.

A.5.9 Termination of an Activity Context (ac)

Pre: none

Post:
$attached_{\mathbf{ac}}(t_2) = \{\}$
$\mathbf{s} \in ExistingSBB(t_2) \Rightarrow \mathbf{ac} \notin activities_{\mathbf{s}}(t_2)$

Description: All SBB entities are detached from the Activity Context. After the termination, no existing SBB entities have the Activity Context in their activities set.

A.5.10 On completion of event routing for an event (e)

Pre: none

Post: $\mathbf{s} \in RootSBB(t_2) \Rightarrow attachment\_count_{\mathbf{s}}(t_2) > 0$

Description: After all eligible SBB entities have received the event, no root SBB entities have an attachment count of zero.

Appendix B Event Router Pseudo Code

Changed in 1.1: This appendix has been updated to include pseudo code for `EventContext.suspendDelivery()`, `EventContext.resumeDelivery()`, *fire event methods which specify a Service ID, and event router handling of suspended Event Contexts.*

The following pseudo code illustrates the desired behavior of the SLEE event router and the methods that interact with the event router. The main goal is to reduce ambiguities that may lead to SLEE implementations that do not implement the specified behavior.

The SLEE vendor is not required to translate the pseudo code directly into implementation code. The SLEE vendor need not follow the same control flow and structure illustrated by the pseudo code, but the SLEE implementation must implement the desired behavior.

Many of the methods illustrated in this appendix have transactional semantics described in Chapter 9. The pseudo code for these methods does not include the necessary code to implement the transactional semantics specified in Chapter 9. The pseudo code in this section also does not show SBB object life cycle method invocations, automatic event subscription, and exception handling.

B.1 Definitions

```
// Conventions:
// SET a set.
// SET[y] is a member of set SET.
// SET[y].ATT is an attribute of SET[y].

// EO[e] is an event object known to the SLEE.
// ET[t] is an event type known to the SLEE.

// ECTX[ctx] is an event context known to the SLEE.
// ECTX[ctx].EO is the event object for the event context.
// ECTX[ctx].ET is the event type for the event.
// ECTX[ctx].AC is the activity context for the event context.
// ECTX[ctx].ADDR is the address passed when the event was fired.
// ECTX[ctx].SERVICEID is the service ID that the event was fired to.
// ECTX[ctx].SUSPENDED is true if the event context is suspended.
// ECTX[ctx].SBBE is the SBB Entity which was being delivered to when the Event Context
//               was suspended

// SVC[v] is a Service.
// A Service has a default event delivery priority and a root SBB.
// SVC[v].PRIORITY is the default event delivery priority of the Service.
// SVC[v].SBB is the root SBB of the Service.
// SVC[v].CHILDOBJ is the set of the child relation object of the Service.
//                This child relation object contains the root SBB entities that have
//                been created for this Service.

// SBB[b] is an SBB component.
// SBB[b].CHILD is the set of child relations of the SBB.
// SBB[b].CHILD[c] is a child relation of the SBB.
// SBB[b].CHILD[c].PRIORITY is the default event delivery priority of the child relation.
// SBB[b].CHILD[c].SBB is the child SBB of the child relation.
// SBB[b].IET is the set of initial event types of the SBB.
// SBB[b].RET is the set of received event types of the SBB.
// SBB[b].MOAET is the set of received event types of the SBB whose mask-on-attach
//              attribute is set to true.
// SBB[b].MOAEN is the set of event names of the received event types of the SBB
//              whose mask on attach attribute is set to true.
// There is a one-to-one mapping between elements of SBB[b].MOAET and the elements
// of SBB[b].MOAEN, i.e. SBB[b].MOAET[x] maps to SBB[b].MOAEN[x] and vice versa.

// SBB[b].IET is subset of SBB[b].RET
// SBB[b].MOAET is subset of SBB[b].RET

// SBB[0] is reserved to represent the SLEE as the logical parent of all root SBBs.
```

```
// This means:
//      SBB[0].CHILD[v].PRIORITY is always the same as SVC[v].PRIORITY
//      SBB[0].CHILD[v].SBB is always the same as SVC[v].SBB

// SBBE[b] is an SBB entity.
// SBBE[b].SBB is the SBB component of the SBB entity.
// SBBE[b].SERVICE is the Service the SBB entity is part of.
// SBBE[b].PARENT is the parent SBB entity of the SBB entity.
// SBBE[b].ROOT is the root SBB entity of the SBB entity's SBB entity tree
// SBBE[b].PRIORITY is the event delivery priority of the SBB entity.
// SBBE[b].CONVNAME is the convergence name of the SBB entity if it is a root SBB entity.
// SBBE[b].CHILDOBJ is the set of child relation objects of the SBB entity
// SBBE[b].CHILDOBJ[c] is a child relation object of the SBB entity.
// SBBE[b].CHILDOBJ[c].CHILD is the child relation of the child relation object.
// SBBE[b].CHILDOBJ[c].SBBE is the set of child SBB entities of a child relation object.
// SBBE[b].CHILDOBJ[c].SBBE[f] is a child SBB entity of a child relation object.
// SBBE[b].ACOUNT is the attachment count of the SBB entity.
// SBBE[b].AC is the set of Activity Contexts that the SBB entity is attached to.
// SBBE[b].AC[a] is an Activity Context that the SBB entity is attached to.

// SBBE[0] is the SLEE as the logical parent SBB entity of all root SBB entities.

// Relationship between SVC[v].CHILDOBJ and SBBE[0].CHILDOBJ[v].
//                 Logically, each Service is represented by a child relation object
//                 in the SLEE SBB entity, i.e. SBBE[0].
//                 SVC[v].CHILDOBJ is the same as SBBE[0].CHILDOBJ[v].

// AC[a] is an Activity Context.
// AC[a].ISSUSPENDED is true if an Event Context for an event fired on the Activity
//                   Context is suspended.
// AC[a].ISRESUMING is true if an Event Context for an event fired on the Activity
//                  Context has been resumed, but the event router has not yet
//                  re-started processing of the event
// AC[a].SBBE is the set of SBB entities attached to the Activity Context.
// AC[a].SBBE[b] is an SBB entity attached to the Activity Context.
// AC[a].DELIVERED is the set of SBB entities that an event has been delivered to.
// AC[a].DELIVERED[a] is an SBB entity that an event has been delivered to.

// A[a] is the Activity object of Activity Context AC[a] and vice versa,
//      i.e. there is a one-to-one mapping between Activity Contexts and
//      Activity objects.

// Each SBB entity attached to an Activity Context has its own event name mask
// and event type mask.
// ENM(SBBE[b],AC[a]) is the set of received event names of SBBE[b].SBB that
//                    SBBE[b] would like to mask from AC[a].
// ETM(SBBE[b],AC[a]) is the set of received event types of SBBE[b].SBB that
//                    SBBE[b] would like to mask from AC[a].

// Q is the SLEE's event queue.
// Q[q] represent an event in the event queue.
// Q[q].ET is the event type of the event.
// Q[q].EO is the event object of the event.
// Q[q].AC is the Activity Context on which the event was fired.
// Q[q].ADDR is the address on which the event was fired.
// Q[q].SERVICEID is the service ID that the event was fired to.
// Q[q].ECTX is the event context for this event.
```

B.2 `ActivityContextInterface` interface `attach` method pseudo code

```
SUBROUTINE ac.attach(sbbe)
       // ac is member of AC.
       // sbbe is member of SBBE.
       if (ac is not a member of sbbe.AC) then
              add sbbe to ac.SBBE
              add ac to sbbe.AC
              ETM(sbbe,ac) = (sbbe.SBB).MOAET
              ENM(sbbe,ac) = (sbbe.SBB).MOAEN
              // If there is an event being routed on this Activity Context, then
```

```
                // the SLEE should deliver this event to this SBB entity.
                if sbbe in ac.DELIVERED then
                        remove sbbe from ac.DELIVERED
                endif
                // Iterate through the SBB entity and the ancestors of the SBB entity.
                do
                        // Increment their attachment count.
                        sbbe.ACOUNT = sbbe.ACOUNT + 1
                        sbbe = sbbe.PARENT
                while sbbe is not SBBE[0]
        endif
END SUBROUTINE
```

B.3 `ActivityContextInterface` interface `detach` method pseudo code

```
SUBROUTINE ac.detach(sbbe)
        // ac is a member of AC.
        // sbbe is a member of SBBE.
        if (ac is a member of sbbe.AC) then
                remove sbbe from ac.SBBE
                remove ac from sbbe.AC
                ETM(sbbe,ac) = {}
                ENM(sbbe,ac) = {}
                // Iterate through the SBB entity and the ancestors of the SBB entity.
                do
                        // Decrement attachment count.
                        sbbe.ACOUNT = sbbe.ACOUNT – 1
                        sbbe = sbbe.PARENT
                while sbbe is not SBBE[0]
        endif
END SUBROUTINE
```

B.4 `SbbContext` interface `getActivities` method pseudo code

```
SUBROUTINE sbbe.getActivities()
        // sbbe is a member of SBBE.
        return sbbe.AC;
END SUBROUTINE
```

B.5 `SbbContext` interface `getEventMask` method pseudo code

```
SUBROUTINE sbbe.getEventMask(ac)
        // sbbe is a member of SBBE.
        // ac is a member of AC.
        return ENM(sbbe,ac);
END SUBROUTINE
```

B.6 `SbbContext` interface `maskEvent` method pseudo code

```
SUBROUTINE sbbe.maskEvent(eventNames[], ac)
        // sbbe is a member of SBBE.
        // eventNames is an array of event names
        // ac is a member of AC.

        // Create a new set for event types.
        met = {}
        for each eventName in eventNames[] do
                et = sbbe.mapEventNameToEventType(eventName)
                add et to met
        endfor
        ETM(sbbe,ac) = met
        ENM(sbbe,ac) = eventNames
END SUBROUTINE
```

B.7 `SbbLocalObject` interface `getSbbPriority` method pseudo code

```
SUBROUTINE sbbe.getPriority()
        // sbbe is a member of SBBE.
        return sbbe.PRIORITY;
END SUBROUTINE
```

B.8 `SbbLocalObject` interface `setSbbPriority` method pseudo code

```
SUBROUTINE sbbe.setPriority(priority)
        // sbbe is a member of SBBE.
        sbbe.PRIORITY = priority;
END SUBROUTINE
```

B.9 `SbbLocalObject` interface `remove` method pseudo code

```
SUBROUTINE sbbe.removeInternal(sbbeParentChildRelationObj)
        // sbbe is a member of SBBE.
        // sbbeParentChildRelationObj is a member of the parent SBB entity's set of
        //                              child relation objects.
        for each childRelationObj in sbbe.CHILDOBJ do
                for each sbbeChild in childRelationObj do
                        sbbeChild.removeInternal(childRelationObj)
                endfor
        endfor
        …
        invoke sbbRemove life cycle method on sbbe
        …
        for each ac in sbbe.AC do
                ac.detach(sbbe)
        endfor
        sbbe.PARENT = null
        remove sbbe from sbbeParentChildRelationObj
        …
        // sbbe can now be reclaimed.
ENDSUBROUTINE

SUBROUTINE sbbe.remove()
        // sbbe is a member of SBBE.
        for each childRelationObj in sbbe.PARENT.CHILDOBJ do
                if sbbe is a member of childRelationObj.SBBE then
                        sbbe.removeInternal(childRelationObj)
                        break
                endif
        endfor
END SUBROUTINE
```

B.10 `ChildRelation` interface `create` method pseudo code

```
SUBROUTINE childRelationObj.create()
        // childRelationObj is a child relation object of the parent SBB
        //                  entity, i.e. it is a member of sbbeParent.CHILDOBJ where
        // sbbeParent is a member of SBBE and is the parent SBB entity that the
        //            child relation object belongs to.
        // childRelationObj.CHILD is the child relation of the child relation object,
        //                        i.e. it is an sbbeParent.CHILDOBJ[c].CHILD and is the
        //                        same as (sbbeParent.SBB).CHILD[c].
        sbbe = new instance of (childRelationObj.CHILD).SBB
        sbbe.SBB = (childRelationObj.CHILD).SBB
        sbbe.PARENT = sbbeParent
       if sbbeParent is SBBE[0] then
                sbbe.ROOT = sbbe
        else
                sbbe.ROOT = sbbeParent.ROOT
        endif
        sbbe.PRIORITY = (childRelationObj.CHILD).PRIORITY
        add sbbe to childRelationObj.SBBE
```

```
        return sbbe
END SUBROUTINE
```

B.11 SBB abstract class fire event methods pseudo code

```
SUBROUTINE sbbe.fire<EventName>(eo, ac, address, serviceID)
        // eo is a member of EO.
        // ac is a member of AC.
        // address is the default address of the event being fired
        // serviceID is the component ID of the service the event is being fired to

        // et is the event type of event being fired, it is typically embedded
        //    in the SLEE generated implementation of this method.
        et = sbbe.mapEventNameToEventType(EventName)
        add (et, eo, ac, address, serviceID) to the tail of the event queue Q
END SUBROUTINE

SUBROUTINE sbbe.fire<EventName>(eo, ac, address)
        sbbe.fire<EventName>(eo, ac, address, null)
END SUBROUTINE
```

B.12 EventContext interface suspendDelivery method pseudo code

```
SUBROUTINE EventContext.suspendDelivery()
        // ectx is the current event context object

        // if already suspended then throw exception
        if ectx.SUSPENDED
                error(“already suspended”)
                return
        endif

        // suspend the event context
        ectx.SUSPENDED = true
        ectx.SBBE = the SBB entity that is being delivered to

        // signal to the event router that it may not process events on the AC
        ectx.AC.ISSUSPENDED = true
        ectx.AC.ISRESUMING = false
END SUBROUTINE
```

B.13 EventContext interface resumeDelivery method pseudo code

```
SUBROUTINE EventContext.resumeDelivery()
        // ectx is the current event context object

        // if already suspended then throw exception
        if ectx.SUSPENDED == false
                error(“not suspended”)
                return
        endif

        // resume the event context
        ectx.SUSPENDED = false

        // signal to the event router that it may process events on the AC
        ectx.AC.ISSUSPENDED = false
        ectx.AC.ISRESUMING = true
END SUBROUTINE
```

B.14 Event router pseudo code

The following pseudo code illustrates the desired behavior of the SLEE event router.

```
SUBROUTINE priorityOfSBBE(sbbe)
        // sbbe is a member of SBBE.
        stack = create empty new stack
```

```
        while sbbe is not SBBE[0] do
                stack.push(sbbe) // push SBBE
                sbbe = sbbe.PARENT
        endwhile
        return stack
ENDSUBROUTINE

SUBROUTINE higherPrioritySBBE(sbbe1, sbbe2)
        // sbbe1 and sbbe2 are members of SBBE.
        stack1 = priorityOfSBBE(sbbe1);
        stack2 = priorityOfSBBE(sbbe2);
        for ever do
                sbbe1a = stack1.pop()
                sbbe2a = stack2.pop()
                if sbbe1a.isIdentical(sbbe2a) then
                        // sbb entities have same ancestor
                        if stack1.isEmpty() then
                                // sbbe1 is an ancestor of sbbe2
                                return sbbe1
                        else if stack2.isEmpty() then
                                // sbbe2 is an ancestor of sbbe1
                                return sbbe2
                        else
                                // do nothing, just keep going down hierarchy
                        endif
                else
                        // sbb entities have different ancestors, so choose based
                        // on higher priority ancestor
                        if sbbe1a.PRIORITY > sbbe2a.PRIORITY then
                                return sbbe1
                        else if sbbe1a.PRIORITY < sbbe2a.PRIORITY then
                                return sbbe2
                        else if identity(sbbe1a) > identity(sbbe2a) then
                                return sbbe1
                        else
                                return sbbe2
                        endif
                endif
        endfor
ENDSUBROUTINE

SUBROUTINE processInitialEvents(svc, q)
        // svc is a member of SVC.
        // q is a member of Q.

        // Compute the one or more convergence names according to the initial-event-select
        // and/or the initial-event-selector-method-name elements of the root SBB of the
        // Service.
        // names is an array of convergence names
        names[] = computeConvergenceNames(…);
        // If initial event selector method indicates this is not an initial event
        // for this Service, then return.
        if names[] is null then
                return
        endif

        // Iterate through each convergence name
        for each name in names[] do
                // Determine if there is a root SBB entity of this Service with
                // this computed convergence name.
                found = false
                for sbbe in (svc.CHILDOBJ).SBBE do
                        if sbbe.CONVNAME == name then
                                found = true
                                break
                        endif
                endfor
                if not found then
                        // No root SBB entity with the convergence name exist.
```

```
                        // Create root SBB entity as child of SLEE
                        sbbe = (svc.CHILDOBJ).create()
                        // If create fails, then process the next convergence name,
                        // don't associate convergence name and don't attach.
                        if create fails then
                                continue
                        endif
                        // Remember the convergence name of this child SBB entity.
                        sbbe.CONVNAME = name
                endif
                // Attach either found or new SBB entity to the Activity Context.
                (q.AC).attach(sbbe);
        endfor
END SUBROUTINE

SUBROUTINE postEventHandlingChecks(sbbe, q)
        // sbbe is the sbb entity that has finished processing an event
        // q is a member of Q.

        // If event is Activity End Event, then detach the SBB entity
        // from the Activity Context.
        if q.ET is Activity Event Event then
                (q.AC).detach(sbbe)
        endif
        // If root SBB entity of SBB entity's SBB entity tree
        // has attachment count of 0, then delete the SBB entity tree
        if (sbbe.ROOT).ACOUNT is 0 then
                remove sbbe.ROOT
        endif
END SUBROUTINE

SUBROUTINE routeEvent(q)
        // q is a member of Q

        if q.ECTX.AC.ISRESUMING
                // now that the event context has been resumed we can do clean up
                postEventHandlingChecks(q.ECTX.SBBE, q)

                // reset the event context suspended states
                q.ECTX.AC.ISSUSPENDED = false
                q.ECTX.AC.ISRESUMING = false
                q.ECTX.ISSUSPENDED = false
                q.ECTX.SBBE = null
        else
                // If the event was fired with a service ID then we only calculate
                // convergence names for that service
                if q.SERVICEID is not null
                        if q.ET is a member of SVC[q.SERVICEID].SBB.IET then
                                processInitialEvents(SVC[q.SERVICEID],q)
                        endif
                else
                        // Iterate through each Service that has the event type as an
                        // initial event type.
                        for each svc in SVC do
                                if q.ET is a member of (svc.SBB).IET then
                                        processInitialEvents(svc,q);
                                endif
                        endfor
                endif
                // Deliver event to SBB entities attached to the Activity Context
                (q.AC).DELIVERED = {}
        endif

        for ever do
                // Find the highest priority SBB entity attached to the
                // Activity Context that the event has not been delivered to.
                highestPrioritySBBE = null;
                for each sbbe in (q.AC).SBBE do
                        if q.SERVICEID is not null and
```

```
                        sbbe.SERVICE != SVC[q.SERVICEID] then
                            continue
                    endif
                    if q.ET is a member of sbbe.SBB.RET and
                       sbbe is not a member of (q.AC).DELIVERED then
                            if highestPrioritySBBE == null then
                                    highestPrioritySBBE = sbbe
                            else
                                    highestPrioritySBBE =
                                            higherPrioritySBBE(sbbe,
                                                    highestPrioritySBBE)
                            endif
                    endif
            endif
            // If no such SBB entity is found, then event routing for this event
            // is complete.
            if highestPrioritySBBE is null then
                    break;
            add highestPrioritySBBE to (q.AC).DELIVERED
            // Check event mask and invoke appropriate event handler method
            if q.ET is not a member ETM(highestPrioritySBBE,q.AC) then
                    invoke event handler method for q.ET on an SBB object
                           that represents highestPrioritySBBE with arguments
                           (q.EO, q.AC)
                    // if the event context was suspended don't do post event handling
                    // checks until the event context has been resumed
                    if q.ECTX.ISSUSPENDED
                            return
                    endif
                    postEventHandlingChecks(sbbe, q);
            endif
        endfor
        // If event is Activity End Event, then detach all remaining SBB entities from the
        // Activity Context.
        if q.ET is Activity Event Event then
                for each sbbe in (q.AC).SBBE do
                        (q.AC).detach(sbbe)
                        // If root SBB entity of SBB entity's SBB entity tree
                        // has attachment count of 0, then delete the SBB entity tree
                        if (sbbe.ROOT).ACOUNT is 0 then
                                remove sbbe.ROOT
                        endif
                endfor
        endif
        remove q from Q
END SUBROUTINE

SUBROUTINE getNextEligibleEvent()
        for each q in Q do
                if q.AC.ISSUSPENDED == false
                        return q
                endif
        endfor
END SUBROUTINE

SUBROUTINE eventRouterThread()
        while (…) do
            q = getNextEligibleEvent()
            routeEvent(q);
       endwhile
END SUBROUTINE
```

Appendix C JCC Resource Adaptor Type

This appendix describes a *recommended* adaptation of a JCC v1.1 resource to the SLEE.

C.1 Resource adaptor type identifier

The resource adaptor type name is "JCC". The resource adaptor type vendor is "javax.csapi.cc.jcc". The resource adaptor type version is "1.1".

C.2 Activity objects

The Activity objects of a JCC resource are the `JccConnection` objects and `JccCall` objects.

C.3 Events

The following table lists the events emitted by a JCC resource and its Activity objects.

Event ***(`JccEvent`s with different JCC event IDs are different event types)***	***Activity Object***		***Remarks***
	JccConnection	***JccCall***	
JCC CALL EVENTS			
CALL_ACTIVE		X	
CALL_CREATED		X	
CALL_EVENT_TRANSMISSION_ENDED		X	1.
CALL_INVALID		X	
CALL_SUPERVISE_END		X	
CALL_SUPERVISE_START		X	
JCC CONNECTION EVENTS			
CONNECTION_ALERTING	X		
CONNECTION_CONNECTED	X		
CONNECTION_CREATED	X		
CONNECTION_DISCONNECTED	X		2.
CONNECTION_FAILED	X		
CONNECTION_ADDRESS_ANALYZE	X		
CONNECTION_ADDRESS_COLLECT	X		
CONNECTION_AUTHORIZE_CALL_ATTEMPT	X		
CONNECTION_CALL_DELIVERY	X		
CONNECTION_MID_CALL	X		

1. After this event there are no more events on the `JccCall` object. The JCC resource adaptor entity responsible for `JccCall` object should indicate to the SLEE that this Activity object has ended. This event should only be delivered to the SLEE after the JCC resource adaptor entity has indicated to the SLEE that all the `JccConnection` Activity objects of the `JccCall` object have also ended.

2. After this event there are no more events on the `JccConnection` object. The JCC resource adaptor entity responsible for the `JccConnection` object should indicate to the SLEE that this Activity object has ended.

C.3.1 Event types

The event type name of an event is the fully qualified name of the constant value field that defines the JCC event ID for the event. For example, the event type name of a `JccConnectionEvent` with a JCC event ID of CONNECTION_ALERTING is "javax.csapi.cc.jcc.JccConnectionEvent.CONNECTION_ALERTING". The event type vendor and event type version must be "javax.csapi.cc.jcc" and "1.1", respectively.

C.3.2 Event classes

The event classes for each of these events are defined by JCC.

- The event class of a CALL_<*XXX*> event is the `javax.csapi.cc.jcc.JccCallEvent` interface.
- The event class of a CONNECTION_<*XXX*> event is the `javax.csapi.cc.jcc.JccConnectionEvent` interface.

C.4 Activity Context Interface Factory interface

The interface of the JCC resource adaptor type specific Activity Context Interface Factory should be as follows:

```
…
import javax.csapi.cc.jcc.JccCall;
import javax.csapi.cc.jcc.JccConnection;
import javax.slee.ActivityContextInterface;
import javax.slee.FactoryException;
import javax.slee.UnrecognizedActivityException;
…
public interface JccActivityContextInterfaceFactory {
        public ActivityContextInterface
                getActivityContextInterface(JccCall call)
                        throws  NullPointerException,
                                UnrecognizedActivityException,
                                FactoryException;
        public ActivityContextInterface
                getActivityContextInterface(JccConnection connection)
                        throws  NullPointerException,
                                UnrecognizedActivityException,
                                FactoryException;
}
```

C.5 Resource adaptor object

The resource adaptor object that is bound to an SBB's component environment when the SBB uses a `resource-adaptor-entity-binding` deployment descriptor element to reference a JCC resource is a `JccProvider` object, i.e. the `resource-adaptor-interface-name` element contains `javax.csapi.cc.jcc.JccProvider`.

C.6 Restrictions

The following restrictions apply:

- Only the SLEE can add itself as a listener. The JCC resource or its resource adaptor should not allow any other objects to add themselves as listeners. An attempt to do so should be rejected by throwing a `SecurityException`.

Appendix D JAIN SIP 1.2 Resource Adaptor Type

This appendix describes a *recommended* adaptation of a JAIN SIP v1.2 resource to the SLEE[51]. JAIN SIP is a Java representation of the Session Initiation Protocol as defined by the IETF in RFC 3261 that also supports SIP extensions, RFC 2976, RFC 3262, RFC 3265, RFC 3311, RFC 3326 and RFC 3248.

For a description of a JAIN SIP 1.1 resource adaptor type please refer to the JAIN SLEE 1.0 specification.

D.1 Introduction to the JAIN SIP 1.2 Resource Adaptor Type

This resource adaptor type provides a SIP API for SLEE applications. It models the transaction and dialog layers of the SIP protocol. The RA type is designed to support multiple classes of SIP applications from one API.

Applications that act as SIP Proxies, Registrars and Redirects typically use the transaction layer exclusively, whilst UAC, UAS and B2BUA applications primarily use the dialog layer.

Events represent SIP Requests or Responses received by the SIP stack (incoming requests and responses) and Timer expiry. Requests with different SIP methods are fired as different event types. Responses with different ranges of response status codes are fired as different event types. Events are fired on transaction or dialog activities.

Applications in the SLEE may create SIP Requests and Responses using the `MessageFactory` and `DialogActivity` objects. SIP Dialogs and Transactions are used to transmit an outgoing Request or Response. Transaction and Dialog activities are used by applications in the SLEE to receive an incoming Request or Response Event.

D.2 Changes since the JAIN SIP 1.1 Resource Adaptor Type

The JAIN SIP 1.2 Resource Adaptor Type adds support for SIP Dialogs. This eases common programming tasks when building UAS, UAC and B2BUA applications in the SLEE. Additionally behavior for SIP Forking is defined.

Event types and activity types from the JAIN SIP 1.1 Resource Adaptor Type are present in the JAIN SIP 1.2 Resource Adaptor Type.

D.3 Resource Adaptor type identifier

The resource adaptor type name is "`JSIP`". The resource adaptor type vendor is "`net.java.slee`". The resource adaptor type version is "`1.2`".

D.4 Activity objects

The Activity objects of a JAIN SIP resource are `ClientTransaction`, `ServerTransaction` and `DialogActivity` objects, i.e. objects that implement the `javax.sip.ClientTransaction`, `javax.sip.ServerTransaction` and `net.java.slee.resource.sip.DialogActivity` interfaces.

The `ClientTransaction` and `ServerTransaction` activities represent event sources for a SIP client and server transactions respectively. Proxies, Registrars and Redirect applications primarily use these activities, as they do not need to use Dialog state. The Dialog activity represents the event source of a particular dialog. This activity is primarily used by UAC, UAS and B2BUA applications, as the application needs to track Dialog state.

[51] JSR 32 – maintenance release

New `ClientTransaction` Activity objects are created by calling the `SipProvider.getNewClientTransaction` method, when sending SIP requests. The `ClientTransaction` Activity ends when a final response is received, or the transaction times out.

`ServerTransaction` Activity objects are created automatically when the resource adaptor receives an incoming SIP request. The activity ends when a final response is sent on the `ServerTransaction`.

`DialogActivity` objects are created by the application calling `SipProvider.getNewDialog` or the `SleeSipProvider.getNewDialog` methods. The `SipProvider.getNewDialog` method is typically invoked when a dialog-creating request (e.g. INVITE, REFER or SUBSCRIBE) is received by the application in order to create UAS Dialog state. The `SleeSipProvider.getNewDialog` methods are typically invoked by the application to create a new UAC dialog.

Dialog activities end when the Dialog transitions to the Terminated state. For more information refer to section D.11.

Note that the `javax.sip.Dialog` object can be typecast to a `net.java.slee.resource.sip.DialogActivity` when used in the SLEE. Note that the `javax.sip.SipProvider` object can be typecast to a `net.java.slee.resource.sip.SleeSipProvider` when used in the SLEE.

D.5 Events

Events represent SIP Requests or Responses received by the SIP stack (incoming requests and responses) and Timer expiry. Requests with different SIP methods are fired as different event types. Responses with different ranges of response status codes are fired as different event types. Different timer expiries are different event types. Events are fired on transaction or dialog activities. There are several event types defined. These are categorized by the Activity each event type is fired on.

D.5.1Events fired on the Client Transaction Activity

Event Type Identifier (name, vendor, version)			Event Class
Name	**Vendor**	**Version**	
javax.sip.message.Response.TRYING	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.PROVISIONAL	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.SUCCESS	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.REDIRECT	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.CLIENT_ERROR	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.SERVER_ERROR	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.GLOBAL_FAILURE	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.Timeout.TRANSACTION	net.java.slee	1.2	`javax.sip.ResponseEvent`

D.5.2Events fired on the Server Transaction Activity

Event Type Identifier (name, vendor, version)			Event Class
Name	**Vendor**	**Version**	
javax.sip.message.Request.INVITE	net.java.slee	1.2	`javax.sip.RequestEvent`

javax.sip.message.Request.ACK	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.message.Request.CANCEL	net.java.slee	1.2	`net.java.slee.resource.sip.CancelRequestEvent`
javax.sip.message.Request.BYE	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.message.Request.OPTIONS	net.java.slee	1.2	`Javax.sip.RequestEvent`
javax.sip.message.Request.REGISTER	net.java.slee	1.2	`Javax.sip.RequestEvent`
javax.sip.message.Request.INFO	net.java.slee	1.2	`Javax.sip.RequestEvent`
javax.sip.message.Request.PRACK	net.java.slee	1.2	`Javax.sip.RequestEvent`
javax.sip.message.Request.UPDATE	net.java.slee	1.2	`Javax.sip.RequestEvent`
javax.sip.message.Request.MESSAGE	net.java.slee	1.2	`Javax.sip.RequestEvent`
javax.sip.message.Request.SUBSCRIBE	net.java.slee	1.2	`Javax.sip.RequestEvent`
javax.sip.message.Request.NOTIFY	net.java.slee	1.2	`Javax.sip.RequestEvent`
javax.sip.message.Request.REFER	net.java.slee	1.2	`Javax.sip.RequestEvent`
javax.sip.message.Request.PUBLISH	net.java.slee	1.2	`Javax.sip.RequestEvent`
javax.sip.message.Request.SIP_EXTENSION	net.java.slee	1.2	`Javax.sip.RequestEvent`

These events represent various SIP messages. The final entry (javax.sip.message.Request.SIP_EXTENSION) is an event type representing a SIP request that is not defined by this Resource Adaptor type.

D.5.3Events fired on Dialog activities

Event Type Identifier (name, vendor, version)			**Event Class**
Name	**Vendor**	**Version**	
javax.sip.Dialog.INVITE	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.Dialog.ACK	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.message.Request.CANCEL	net.java.slee	1.2	`net.java.slee.resource.sip.CancelRequestEvent`
javax.sip.Dialog.BYE	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.Dialog.OPTIONS	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.Dialog.REGISTER	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.Dialog.INFO	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.Dialog.PRACK	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.Dialog.UPDATE	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.Dialog.MESSAGE	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.Dialog.SUBSCRIBE	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.Dialog.NOTIFY	net.java.slee	1.2	`javax.sip.RequestEvent`

javax.sip.Dialog.REFER	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.Dialog.PUBLISH	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.Dialog.SIP_EXTENSION	net.java.slee	1.2	`javax.sip.RequestEvent`
javax.sip.message.Response.TRYING	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.PROVISIONAL	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.SUCCESS	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.REDIRECT	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.CLIENT_ERROR	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.SERVER_ERROR	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.message.Response.GLOBAL_FAILURE	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.Timeout.TRANSACTION	net.java.slee	1.2	`javax.sip.ResponseEvent`
javax.sip.Timeout.Dialog	net.java.slee	1.2	`net.java.slee.resource.sip.DialogTimeoutEvent`
javax.sip.Dialog.FORKED	net.java.slee	1.2	`net.java.slee.resource.sip.DialogForkedEvent`

Events fired on Dialog activities are said to be “in-dialog”. These events represent various SIP messages. The javax.sip.Dialog. SIP_EXTENSION event type represents a Request that does not have an event type defined by this Resource Adaptor type.

For more details related to the semantics of SIP Dialog and associated events, please refer to sections D.6, D.9, D.11.1, D.11.2, D.11.3, D.12 and D.14.

D.6 Mapping of Events to Activities

Events represent incoming SIP protocol messages. An incoming SIP protocol message can be either a Request (if the application is a server for that SIP transaction) or a Response (if the application is the client for that SIP transaction).

Events are either in-dialog or out-of-dialog. An in-dialog event is fired on a Dialog activity. An out-of-dialog event is fired on a transaction activity. An event is in-dialog if the incoming Request or incoming Response has a dialog ID that matches an existing Dialog activity. An event is out-of-dialog if the incoming Request or incoming Response does not contain a dialog ID, or if the dialog ID it contains does not match an existing Dialog activity.

If an application acts as a client for a particular SIP transaction and the application does not use the Dialog layer, then it should use the `SipProvider.getNewClientTransaction` method, attach to the `ClientTransaction` activity to receive any responses, and send the Request. The response (if any) to the request will be fired on the `ClientTransaction` activity. If there is no response a timeout event is fired on the `ClientTransaction` activity.

If an application acts as a client for a particular SIP transaction and the application uses the Dialog layer, then it should use the `Dialog.sendRequest(Request)` method. The response (if any) to the request will be fired on the Dialog’s activity.

D.7 Activity Context Interface Factory interface

The interface of the JAIN SIP resource adaptor type specific Activity Context Interface Factory is defined as follows:

The JAIN SIP Activity Context Interface Factory is defined as follows:

```
package net.java.slee.resource.sip;

import javax.slee.ActivityContextInterface;
import javax.slee.FactoryException;
import javax.slee.UnrecognizedActivityException;

import javax.sip.ClientTransaction;
import javax.sip.ServerTransaction;
import javax.sip.Dialog;

public interface SipActivityContextInterfaceFactory {
        public ActivityContextInterface
                getActivityContextInterface(ClientTransaction clientTransaction)
                        throws  UnrecognizedActivityException,
                                FactoryException;

        public ActivityContextInterface
                getActivityContextInterface(ServerTransaction serverTransaction)
                        throws  UnrecognizedActivityException,
                                FactoryException;

        public ActivityContextInterface
                getActivityContextInterface(DialogActivity dialog)
                        throws  UnrecognizedActivityException,
                                FactoryException;
}
```

D.8 Resource Adaptor interface object

The JAIN SIP Resource Adaptor SBB Interface provides SBBs with access to the JAIN SIP objects required for creating and sending messages. It is defined as follows:

```
package net.java.slee.resource.sip;

import javax.sip.SipProvider;
import javax.sip.address.AddressFactory;
import javas.sip.address.Address;
import javax.sip.header.HeaderFactory;
import javax.sip.message.MessageFactory;

public interface SleeSipProvider extends SipProvider {
    public AddressFactory getAddressFactory();
    public HeaderFactory getHeaderFactory();
    public MessageFactory getMessageFactory();

    public DialogActivity getNewDialog(Address from, Address to) throws SipException;
    public DialogActivity getNewDialog(DialogActivity incomingDialog,
                                       boolean useSameCallId) throws SipException;

    public boolean isLocalSipURI(SipURI uri);
    public boolean isLocalHostname(String host);
    public SipURI getLocalSipURI(String transport);
    public ViaHeader getLocalVia(String transport, String branch)
                        throws TransportNotSupportedException;

    public DialogActivity forwardForkedResponse(ServerTransaction origServerTransaction,
                                                Response response)throws SipException;

    public boolean acceptCancel(CancelRequestEvent cancelEvent,
                                boolean isProxy);
}
```

The `getAddressFactory` method:

This method returns an implementation of the `AddressFactory` interface.

The `getHeaderFactory` method:

This method returns an implementation of the `HeaderFactory` interface.

The `getMessageFactory` method:

This method returns an implementation of the `MessageFactory` interface.

The `getNewDialog(Address,Address)` method:

This method creates a new SIP Dialog, starts a new Activity in the SLEE, and returns the new `DialogActivity`. The new Dialog is a UAC Dialog. The parameters specify the local and remote addresses for the Dialog.

This method takes the following arguments:

`from` – This specifies the local address for the Dialog.

`to` – This specifies the remote address for the Dialog.

This method throws a `SipException` if the new Dialog is unable to be created.

The `getNewDialog(DialogActivity, boolean)` method:

This method creates a new SIP Dialog, starts a new Activity in the SLEE, and returns the new `DialogActivity`. The new Dialog is a UAC Dialog. This method copies the local and remote addresses from incoming Dialog to the newly created Dialog. It generates a new local tag and the remote tag will be null. If the `useSameCallId` parameter is true then the newly created Dialog will use the Call ID from the incoming Dialog, otherwise a new Call ID is generated.

This method takes the following arguments:

`incomingActivity` – This specifies the incoming Dialog which is used to copy headers from.

`useSameCallId` – This specifies whether or not the newly created Dialog uses the same call ID as the incoming Dialog.

The `isLocalSipURI` method:

This method returns true if the URI is local to the SIP stack.

The `isLocalHostname` method:

This method returns true if the hostname is local to the SIP stack.

The `getLocalVia` method:

This method returns a via header with the correct local address for the SIP stack.

This method takes the following arguments:

`transport` - a case in-sensitive transport name, such as “udp”, “tcp”, or “tls”. This value is used as the result of the `getTransport` method on `javax.sip.header.ViaHeader`.

`branch` – the branch for the via header, if null the RA will automatically generate a valid branch parameter.

This method throws a `TransportNotSupportedException` if the given transport is not supported by the Resource Adaptor.

The `acceptCancel(CancelRequestEvent cancelEvent, boolean isProxy)` method:

This method is a convenience method for handling CANCEL requests. When a CANCEL arrives, and application can ask the RA to handle the request by calling this method. The behavior of the RA is dependent on whether the CANCEL matched the INVITE, and also whether the application is acting as a proxy or not, as determined by the `isProxy` parameter. The method returns true if the CANCEL matched an INVITE. The behavior of the method is summarized as follows.

	isProxy=false	**isProxy=true**
CANCEL matched INVITE	Send “200 OK” response to CANCEL. Send “487 Request Terminated” response to INVITE.	Send “200 OK” response to CANCEL. The Proxy application must cancel outstanding branches and wait for responses (RFC3261 §16.10).
CANCEL did not match INVITE	Send “481 Call or Transaction Does Not Exist” response to CANCEL.	Do nothing. Proxy application must statelessly forward the CANCEL downstream (RFC3261 §16.10).

The resource adaptor is expected to check if the CANCEL request was processed by any SBBs. If the CANCEL request was not processed by an SBB, the RA is expected to send a “481 Call or Transaction Does Not Exist”.

This method takes the following arguments:

`cancelEvent` - contains the CANCEL request, its Server Transaction and the matching INVITE Server Transaction, if any.

`isProxy` - specifies if the application is acting as a SIP Proxy.

This method returns `true` if the CANCEL matched an INVITE, otherwise `false`.

For more information on CANCEL handling refer to section D.13.

The `forwardForkedResponse(ServerTransaction origServerTransaction, Response response)` method:

This method is used when forwarding a forked response, that was received in a `DialogForkedEvent`. A transparent B2BUA application must use this method to forward forked responses upstream. This method sends the response on the supplied `javax.sip.ServerTransaction`, and creates a new Dialog Activity that the SBB should attach to, to receive mid-dialog requests on the new dialog.

Sending a forked response upstream using the `sendResponse(javax.sip.message.Response)` method on `javax.sip.ServerTransaction` will not work correctly, because if the caller sends any mid-dialog requests on the new dialog, the RA Entity will not match them with a dialog activity, and will not fire these request events on the Dialog Activity.

If the response is a final response, all other early dialogs will be terminated.

This method takes the following arguments:

`origServerTransaction` - the SIP transaction used to send the response on. This is typically obtained using the `getAssociatedServerTransaction` method on the `DialogActivity` interface during processing of the `DialogForkedEvent`.

`response` - the `javax.sip.message.Response` that was received in a `DialogForkedEvent`. This parameter is copied, and the copy is modified so that it is suitable to send on the server transaction, i.e. the Via, CSeq, From, and To headers will match the upstream call leg, as per the `createResponse(javax.sip.ServerTransaction, javax.sip.message.Response)` method. The parameter is not modified.

This method returns a new `DialogActivity` resulting from this forked response. Applications should attach to this activity to receive any mid-dialog requests.

A `SipException` is thrown if the RA is unable to create the forked dialog, or send the response.

D.9 The `DialogActivity` interface

The `DialogActivity` is the Java type for an Activity in the SLEE that represents a SIP Dialog. Note that JAIN SIP 1.2 defines the interface `javax.sip.Dialog`. When running in the SLEE `javax.sip.Dialog` may be typecast to `net.java.slee.resource.sip.DialogActivity`. The interface for the Dialog activity is defined as follows:

```
package net.java.slee.resource.sip;

import javax.slee.ActivityContextInterface;
import javax.slee.FactoryException;
import javax.slee.UnrecognizedActivityException;

import javax.sip.ClientTransaction;
import javax.sip.Dialog;
import javax.sip.TransactionUnavailableException;
import javax.sip.SipException;
import javax.sip.message.Request;
import javax.sip.message.Response;

public interface DialogActivity extends Dialog {
        public ClientTransaction sendRequest(Request request)
                throws SipException;
        public Request createRequest(Request origRequest)
                throws SipException;
        public Response createResponse(ServerTransaction origServerTransaction,
                        Response receivedResponse) throws SipException;

        public void associateServerTransaction(ClientTransaction ct,
                         ServerTransaction st);
        public ServerTransaction getAssociatedServerTransaction(ClientTransaction ct);
        public ClientTransaction sendCancel() throws SipException;
}
```

The `sendRequest` method

The `sendRequest` method is used to send a request to the Dialog's remote party on a new Client Transaction. Any response to this request will be fired on the Activity associated with this Dialog. This method should be used by an application that is Dialog stateful, in preference to the `javax.sip.Dialog.sendRequest(ClientTransaction)` method. The reason for this preference is that the API is simpler and SLEE resources are used.

The method returns the Client Transaction used to represent the SIP transaction. Note that the returned Client Transaction does not have a corresponding Activity in the SLEE.

This method will set the From and the To tags for the outgoing request. This method increments the dialog sequence number and sets the correct sequence number to the outgoing Request and associates the client transaction with this dialog. Note that any tags assigned by the user will be over-written by this method. If the caller sets no `RouteHeader` in the Request to be sent out, the implementation of this method will add the `RouteHeader` from the route set that is maintained in the dialog. If the caller sets the route header, the implementation will leave the route headers unaltered. This allows the application to manage its own route set if so desired.

This method throws the following exception:

- `SipException` This exception is thrown if the Request is unable to be sent for any reason.

The `createRequest` method

The `createRequest` method is used to create a Request that will be sent on this Dialog, where the Request has been received on an incoming Dialog. This scenario is common for B2BUA applications. The method returns a new Request to be sent on the Dialog. The method copies the headers and body present in the argument to the return result, and sets the headers in the return result to reflect that the message is to be sent on this Dialog.

This method takes the following argument:

`origRequest` – This specifies the incoming Request that will be used as part of the process of creating a new Request.

This method throws a `SipException` if a new Request is unable to be created.

The `createResponse` method

The `createResponse` method is used to create a Response that is to be forwarded on this Dialog. This is typically performed when a B2BUA application receives a response from one Dialog and wishes to forward the response on another Dialog. The method returns a new Response to be sent on this Dialog. The method copies the headers and body present in the response argument to the return result, and copies the transactions' identifier to the return result. This method is intended to be used in conjunction with the `associateServerTransaction` and `getAssociatedServerTransaction` methods.

This method takes the following arguments:

`origServerTransaction` – This represents the `ServerTransaction` on the Dialog which the newly created response will be sent on.

`receivedResponse` - This is the Response which will be copied as part of the process of creating the new Response.

This method throws a `SipException` if it is unable to create a new Response.

The `associateServerTransaction` method

This method creates an association between a `ServerTransaction` from another Dialog with a `ClientTransaction` for use with this Dialog. This association can be accessed later via the `getAssociatedServerTransaction` method. The association is cleared when the `ServerTransaction` terminates. It is used primarily by B2BUA applications that wish to forward responses from a UAC Dialog to a UAS Dialog. It is intended to be used in conjunction with the `getAssociatedServerTransaction` and `createResponse(ServerTransaction, Response)` methods.

This method takes the following arguments:

`clientTransaction` – This argument represents a Client Transaction for this Dialog.

`serverTransaction` – This argument represents a Server Transaction for another Dialog.

The `getAssociatedServerTransaction` method

This method returns the `ServerTransaction` associated with the specified `ClientTransaction`. If no association exists for the specified `ClientTransaction` then this method returns `null`. This method is intended to be used in conjunction with the `associateServerTransaction` and `createResponse(ServerTransaction, Response)` methods.

This method takes the following argument:

`clientTransaction` – This argument represents a Client Transaction for this Dialog.

The `sendCancel` method

This method creates and sends a CANCEL request for the last INVITE request that was sent on this dialog. This method may be used for cancelling initial INVITEs or re-INVITEs. This method does not directly affect the dialog state. The CANCEL should cause the downstream server to send a 487 response to the INVITE, which will automatically end the dialog if the INVITE was the initial request.

This method returns a `javax.sip.ClientTransaction` activity object, representing the CANCEL's client transaction. The SBB may obtain the `javax.slee.ActivityContextInterface` for this activity and attach to it, if it is interested in the response to the CANCEL. Otherwise, this activity may be safely ignored.

This method throws a `SipException` if there was no active INVITE client transaction to cancel, or * if there was an error sending the CANCEL request.

D.10 Restrictions

The following restrictions apply:

- The resource adaptor implementation should prevent SBBs from adding themselves as SIP listeners, or changing the SIP network configuration. Any attempt to do so should be rejected by throwing a `SecurityException`.
- The `javax.sip.Dialog.setApplicationData`, `javax.sip.Dialog.getApplicationData`, `javax.sip.Transaction.setApplicationData`, and `javax.sip.Transaction.getApplicationData` methods are not to be used by SLEE applications, instead SBB entity CMP state or Activity Context Interface CMP state should be used. Any attempt to invoke these methods should be rejected by throwing a `SecurityException`.

D.11 SIP Application Scenarios

There are several common scenarios for SIP applications. This section provides illustrative examples for several of these scenarios.

D.11.1UAC in SLEE, UAS external

A UAC application can be written in SLEE using this RA type. Such a UAC can be represented by an SBB. This SBB uses the dialog APIs. It does not need to use the transaction activities as all in-dialog messages are fired on the dialog activity. There is one SBB entity per dialog. This SBB entity creates a SIP Dialog, attaches to the dialog's activity, and sends an INVITE. As it is attached to the Dialog activity it will receive any responses or requests that are fired on the Dialog activity. That is it will receive the incoming OK corresponding to the INVITE it sent, and it will receive any incoming requests that are fired on the Dialog activity, i.e. if the UAS sends a BYE, the event representing the BYE will be delivered to the SBB.

```
package com.acme.example;

import javax.slee.Sbb;
import javax.slee.SbbLocalObject;
import net.java.slee.resource.sip.DialogActivity;
import javax.sip.*;
import javax.sip.address.Address;
import javax.sip.message.*;
import javax.sip.header.*;
import javax.slee.ActivityContextInterface;

public abstract class SimpleUACSbb implements Sbb{

    public void createDialog(Address from, Address to, ContactHeader contact)
                throws SipException{

        // create the dialog activity
        DialogActivity dialog = getSipProvider().getNewDialog(from, to);
        ActivityContextInterface dialogACI =
            getSipACIFactory().getActivityContextInterface(dialog);
```

```
        // attach to its ACI so that the SBB entity receives messages
        dialogACI.attach(myLocalObject);

        // create and send an INVITE
        Request invite = dialog.createRequest(Request.INVITE);
        invite.setHeader(contact);
        dialog.sendRequest(invite);
    }

    // handle 2xx OK response
    public void on2xxResponse(ResponseEvent event, ActivityContextInterface aci) {
        try{
            // send ACK
            DialogActivity dialog = (DialogActivity) aci.getActivity();
            CSeqHeader cseq =
                (CSeqHeader)event.getResponse().getHeader(CSeqHeader.NAME);
            Request ack = dialog.createAck(cseq.getSeqNumber());
            dialog.sendAck(ack);
        }catch(SipException e){
            //exception handling code
        }
    }

    // other request handling methods
    // lifecycle methods

    private SbbLocalObject myLocalObject;
}
```

D.11.2UAS in SLEE, UAC external

A UAS application can be written in SLEE using this RA type. Such a UAS can be represented by an SBB. This SBB uses the transaction and dialog APIs. It uses one transaction activity and one dialog activity.

There is one SBB entity per dialog. This SBB entity is created when an INVITE is received, as INVITE is its initial event. As the INVITE message does not contain a full dialog ID, it is fired on a `ServerTransaction` activity. On receipt of the INVITE the SBB entity creates a Dialog and attaches to the Dialog activity. It then sends the OK using the `ServerTransaction`. The ACK is then fired on the Dialog activity as it represents a full dialog.

If the UAS later decides to end the dialog it creates a new client transaction via the Dialog, i.e. it uses the `Dialog.sendRequest(req)` method.

```
package com.acme.example;

import javax.slee.Sbb;
import javax.slee.SbbLocalObject;
import net.java.slee.resource.sip.DialogActivity;
import javax.sip.*;
import javax.sip.address.Address;
import javax.sip.message.*;
import javax.sip.header.*;
import javax.slee.ActivityContextInterface;

public abstract class SimpleUASSbb implements Sbb{

    // Initial request
    public void onInitialInvite(RequestEvent event, ActivityContextInterface aci) {
        // ACI is the server transaction activity
        ServerTransaction st = event.getServerTransaction();
        try {
            // Create the dialog representing the incoming call legs.
```

```
            DialogActivity incomingDialog =
                (DialogActivity) getSipProvider().getNewDialog(st);

          // Obtain the dialog activity context and attach to it
            ActivityContextInterface incomingDialogACI =
                getSipACIFactory().getActivityContextInterface(incomingDialog);
            incomingDialogACI.attach(myLocalObject);
        } catch (SipException e) {
            // exception handling code
        }
    }

    // other request handling methods
    // lifecycle methods

    private SbbLocalObject myLocalObject;
}
```

D.11.3B2BUA in SLEE, with an UAC and UAS external

A B2BUA is an application that acts as both a UAC and UAS. In this scenario an external UAC initiates a call, and an external UAS terminates the call. This B2BUA sample application shows how to forward messages between the UAC and UAS dialogs.

```
package com.acme.example;

import javax.slee.Sbb;
import javax.slee.SbbLocalObject;
import net.java.slee.resource.sip.DialogActivity;
import javax.sip.*;
import javax.sip.address.Address;
import javax.sip.message.*;
import javax.sip.header.*;
import javax.slee.ActivityContextInterface;

public abstract class SimpleB2BUASbb implements Sbb{
    // Initial request
    public void onInitialInvite(RequestEvent event, ActivityContextInterface aci) {
        // ACI is the server transaction activity
        ServerTransaction st = event.getServerTransaction();
      try {
           // Create the dialogs representing the incoming and outgoing call legs.
            DialogActivity incomingDialog
                = (DialogActivity) getSipProvider().getNewDialog(st);
           DialogActivity outgoingDialog
                = getSipProvider().getNewDialog(incomingDialog, true);

          // Obtain the dialog activity contexts and attach to them
            ActivityContextInterface outgoingDialogACI
                = getSipACIFactory().getActivityContextInterface(outgoingDialog);
           ActivityContextInterface incomingDialogACI
                = getSipACIFactory().getActivityContextInterface(incomingDialog);

            incomingDialogACI.attach(myLocalObject);
           outgoingDialogACI.attach(myLocalObject);

          // Record which dialog is which, so we can find the peer dialog
           // when forwarding messages between dialogs.
           setIncomingDialog(incomingDialogACI);
           setOutgoingDialog(outgoingDialogACI);

            forwardRequest(st, outgoingDialog);
        } catch (SipException e) {
            sendErrorResponse(st, Response.SERVICE_UNAVAILABLE);
        }
```

```
    }

   // Responses
   public void on1xxResponse(ResponseEvent event, ActivityContextInterface aci) {
       processResponse(event, aci);
    }

   public void on2xxResponse(ResponseEvent event, ActivityContextInterface aci) {
       processResponse(event, aci);
    }

     // other responses handled the same way as above

   // Mid-dialog requests
    public void onAck(RequestEvent event, ActivityContextInterface aci) {
       processMidDialogRequest(event, aci);
    }

     public void onBye(RequestEvent event, ActivityContextInterface aci) {
       processMidDialogRequest(event, aci);
    }

    // Other mid-dialog requests handled the same way as above


   // Helpers
   private void processMidDialogRequest(RequestEvent event,
                                        ActivityContextInterface dialogACI) {
        try {
            // Find the dialog to forward the request on
            ActivityContextInterface peerACI = getPeerDialog(dialogACI);
            forwardRequest(event.getServerTransaction(),
                           (DialogActivity)peerACI.getActivity());
        } catch (SipException e) {
            sendErrorResponse(event.getServerTransaction(),
                              Response.SERVICE_UNAVAILABLE);
        }
    }

   private void processResponse(ResponseEvent event, ActivityContextInterface aci){
        try {
            // Find the dialog to forward the response on
            ActivityContextInterface peerACI = getPeerDialog(aci);
            forwardResponse((DialogActivity)aci.getActivity(),
                            (DialogActivity)peerACI.getActivity(),
                            event.getClientTransaction(),
                            event.getResponse());
        } catch (SipException e) {
             // exception handling code
        }
    }

   private ActivityContextInterface getPeerDialog(ActivityContextInterface aci)
                                        throws SipException {
        if (aci.equals(getIncomingDialog())) return getOutgoingDialog();
        if (aci.equals(getOutgoingDialog())) return getIncomingDialog();
        throw new SipException("could not find peer dialog");
    }

   private void forwardRequest(ServerTransaction st, DialogActivity out)
                       throws SipException {

         // Copies the request, setting the appropriate headers for the dialog.
Request incomingRequest = st.getRequest();
        Request outgoingRequest = out.createRequest(incomingRequest);

        if (incomingRequest.getMethod().equals(Request.ACK)) {
            // Just forward the ACK statelessly - don't need to remember
             // transaction state
```

```
            out.sendAck(outgoingRequest);
        }
        else {
            // Send the request on the dialog activity
            ClientTransaction ct = out.sendRequest(outgoingRequest);

            // Record an association with the original server transaction,
             // so we can retrieve it when forwarding the response.
            out.associateServerTransaction(ct, st);
        }
    }

   private void forwardResponse(DialogActivity in,
                                DialogActivity out,
                                ClientTransaction ct,
                                Response receivedResponse) throws SipException {
        // Find the original server transaction that this response
         // should be forwarded on.
        ServerTransaction st = in.getAssociatedServerTransaction(ct);

         // could be null
        if (st == null)
             throw new SipException("could not find associated server transaction");

         // Copy the response across, setting the appropriate headers for the dialog
        Response outgoingResponse = out.createResponse(st, receivedResponse);

         // Forward response upstream.
        try {
            st.sendResponse(outgoingResponse);
        } catch (InvalidArgumentException e) {
            throw new SipException("invalid response", e);
        }
    }

   private void sendErrorResponse(ServerTransaction st, int statusCode) {
        try {
            Response response = getSipProvider().getMessageFactory().createResponse(
                                 statusCode, st.getRequest());
            st.sendResponse(response);
        } catch (Exception e) {
            // exception handling code
        }
    }

    // other request handling methods
    // lifecycle methods

    // CMP field accessors for each Dialogs ACI
   public abstract void setIncomingDialog(ActivityContextInterface aci);
   public abstract ActivityContextInterface getIncomingDialog();
   public abstract void setOutgoingDialog(ActivityContextInterface aci);
   public abstract ActivityContextInterface getOutgoingDialog();

    private SbbLocalObject myLocalObject;
}
```

D.11.4Proxy in SLEE, UAC and UAS external

A Proxy can be written in SLEE using this RA type. Such a Proxy can be represented by an SBB. This SBB uses the transaction layer and transaction activities exclusively. It does not use the SIP Dialog layer. There is one SBB entity per client transaction and server transaction pair. The Proxy SBB entity is instantiated when a request is received on a server transaction. It then creates a client transaction, attaches to the client transaction activity, clones the incoming request forming a new request, modifies the new requests headers as appropriate, and transmits the new request .

D.12 State machines for Dialogs

The SIP protocol specifies a Finite State Machine for a Dialog. This section describes the expected behavior of a Resource Adaptor that implements the JAIN SIP 1.2 RA Type. The Dialog state machine is described in RFC 4235 and is reproduced for convenience in Figure 25.

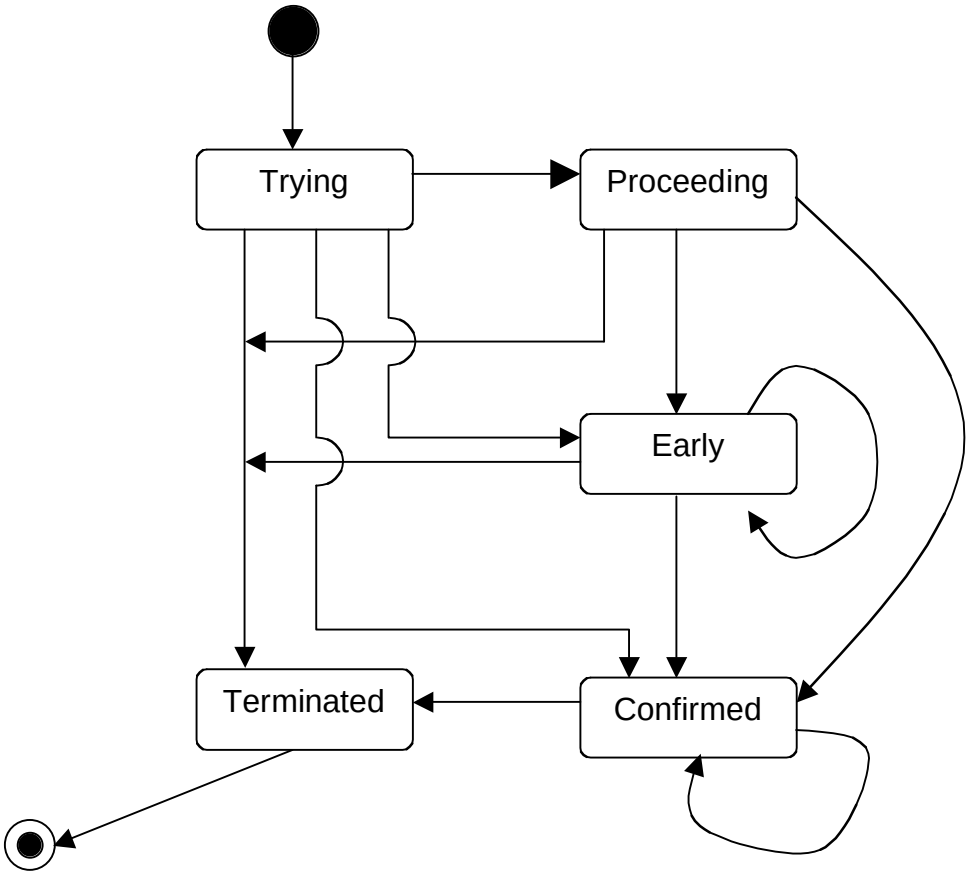

Figure 25 JAIN SIP RA type and Dialog FSM

D.12.1 UAC Dialog FSM and JAIN SIP RA Type

Table 2 describes the logic of the Resource Adaptor for the UAC scenario.

Current Dialog State	Input	Action(s)	Next Dialog State
Does not exist	Application requests RA to create Dialog state by invoking `SipProvider.getNewDialog(ClientTransaction)`, or `SleeSipProvider.getNewDialog()`.	RA creates dialog state. RA transitions dialog state. RA starts dialog activity.	Trying

Trying	Receipt of response 1xx-no tag.	RA transitions Dialog state. RA fires event on Dialog activity.	Proceeding
Trying, or Proceeding	Receipt of response 1xx-with tag	RA transitions Dialog state. RA fires event on Dialog activity.	Early
Trying, or Proceeding, or Early	Receipt of response 2xx	RA transitions Dialog state. RA fires event on Dialog activity.	Confirmed
Trying, or Proceeding, or Early	Receipt of error response or transaction timeout for dialog creating transaction	RA transitions Dialog state. RA fires event on Dialog activity. RA ends dialog activity.	Terminated
Early	Application is client for a mid dialog transaction and any response (error or non error) is received	RA fires event on dialog activity.	Early
Early	Application is client for a transaction and a local transaction timeout is generated by the SIP stack	RA fires appropriate event (Transaction Timeout) on dialog activity.	Early
Early	Receipt of 1xx or 2xx response with a different tag	RA creates a new Dialog in the Early state. RA fires DialogForkedEvent on the original dialog.	Early
Confirmed	Application is client for a transaction and any response (error or non-error) is received	RA fires event on Dialog activity.	Confirmed
Confirmed	Receipt of a 2xx response with a different to tag (differing from the to-tag in the original 2xx response).	RA generates and transmits an ACK. RA transmits BYE on a new client transaction.	Confirmed
Confirmed	Application is client for a transaction and a local transaction timeout is generated by the SIP stack	RA fires appropriate event (Transaction Timeout) on dialog activity.	Confirmed
Trying, Proceeding, Early or Confirmed	Application invokes Dialog.delete() method	RA transitions Dialog state. RA ends activity.	Terminated
Confirmed	Response to a Dialog-terminating request arrives. (e.g. a response to a BYE request arrives.)	RA fires appropriate event on Dialog activity. RA transitions Dialog state. RA ends dialog activity. RA transmits request.	Terminated
Confirmed	Response to a of Dialog-terminating request is sent. (e.g. a response to a BYE request is sent.)	RA transitions Dialog state. RA fires appropriate event on Dialog activity. RA ends dialog activity.	Terminated
Confirmed	Client transaction times out awaiting response to a Dialog terminating request. E.g. a response to a BYE request does not occur within the client transactions timer period.	RA fires transaction timeout event on Dialog activity. RA transitions Dialog state. RA ends Dialog activity.	Terminated

Confirmed	Server transaction times out awaiting response from an application to a Dialog terminating request. E.g. a response to a BYE request is not generated by the application in the SLEE within the server transactions timer period.	RA transitions Dialog state. RA ends Dialog activity.	Terminated
Confirmed	RA determines that Dialog is invalid (e.g. using SIP Session Timers).	RA transitions Dialog state. RA fires javax.sip.Dialog.TIMEOUT vent on Dialog activity. RA ends Dialog activity.	Terminated

Table 2 RA Type behaviour for `DialogActivity` in the UAC scenario

D.12.2 UAS Dialog FSM and JAIN SIP RA Type

Table 3 describes the logic of the Resource Adaptor for the UAS scenario.

Current Dialog State	Input	Action(s)	Next Dialog State
Does not exist	Receipt of a SIP Request that may create a dialog (e.g. INVITE) that does not match an existing dialog	RA starts a `ServerTransaction` activity and fires the request event on the activity. Application handles the event and requests RA to create Dialog state by invoking SIPProvider.getNewDialog(`ServerTransaction`) RA creates dialog state. RA transitions dialog state. RA starts dialog activity.	Trying.
Does not exist	Receipt of a SIP Request that may create a dialog (e.g. INVITE) that does not match an existing Dialog	RA starts a `ServerTransaction` activity and fires the request event on the activity. Application handles the event and does not request the RA to create Dialog state.	Does not exist.
Original UAS Dialog in Early state	Application calls `SleeSipProvider.forwardForkedResponse()` with 1xx response	RA creates a new UAS Dialog, sets its state to the Early state and starts a new Dialog activity. There is no change to the original UAS Dialog.	Original UAS Dialog in Early state, New Dialog in Early state.
Original UAS Dialog in Early state	Application calls `SleeSipProvider.forwardForkedResponse()` with 2xx response	RA creates a new UAS Dialog, sets its state to the Confirmed state, and starts a new Dialog activity. The original dialog, and any forked dialogs are Terminated.	Original Dialog Terminated, any forked Dialogs Terminated. New Dialog in Confirmed state.
Trying	Application requests transmission of response 1xx-no tag on initial `ServerTransaction`	RA transitions Dialog state. RA transmits the response.	Proceeding
Trying, or Proceeding	Application requests transmission of response 1xx-with tag on initial `ServerTransac-`	RA transitions Dialog state. RA transmits the response.	Early

	`tion`		
Trying, or Proceeding, or Early	Application requests transmission of response 2xx on initial `ServerTransaction`	RA transitions Dialog state. RA ends `ServerTransaction` activity. RA transmits the response.	Confirmed
Trying, or Proceeding, or Early	Application requests transmission of error response on initial `ServerTransaction`	RA transitions Dialog state. RA ends Dialog and `ServerTransaction` activities. RA transmits the response.	Terminated
Early	Application is client for a mid dialog transaction and any response (error or non error) is received	RA transitions Dialog state. RA fires event on dialog activity.	Early
Early	Application is client for a transaction and a local transaction timeout is generated by the SIP stack	RA fires appropriate event (Transaction Timeout) on dialog activity.	Early
Confirmed	Application is client for a transaction and any response (error or non-error) is received	RA fires event on Dialog activity.	Confirmed
Confirmed	Application is client for a transaction and a local transaction timeout is generated by the SIP stack	RA fires appropriate event (Transaction Timeout) on dialog activity.	Confirmed
Trying, Proceeding, Early or Confirmed	Application invokes Dialog.delete() method	RA transitions Dialog state. RA ends activity.	Terminated
Confirmed	Response to a Dialog-terminating request arrives. (e.g. a response to a BYE request arrives.)	RA fires appropriate event on Dialog activity. RA transitions Dialog state. RA ends dialog activity. RA transmits request.	Terminated
Confirmed	Response to a of Dialog-terminating request is sent. (e.g. a response to a BYE request is sent.)	RA transitions Dialog state. RA fires appropriate event on Dialog activity. RA ends dialog activity.	Terminated
Confirmed	Client transaction times out awaiting response to a Dialog terminating request. E.g. a response to a BYE request does not occur within the client transactions timer period.	RA fires transaction timeout event on Dialog activity. RA transitions Dialog state. RA ends Dialog activity.	Terminated
Confirmed	Server transaction times out awaiting response from an application to a Dialog terminating request. E.g. a response to a BYE request is not generated by the application in the SLEE within the server transactions timer period.	RA transitions Dialog state. RA ends Dialog activity.	Terminated
Confirmed	RA determines that Dialog is invalid (e.g. using SIP Session Timers).	RA transitions Dialog state. RA fires javax.sip.Dialog.TIMEOUT vent on Dialog activity. RA ends Dialog activity.	Terminated

Table 3 RA Type behaviour for `DialogActivity` in the UAS scenario

D.12.3 Dialog initiating and Dialog terminating requests

Tables Table 2 and Table 3 describe “dialog initiating” and “dialog terminating” requests.

Dialog Activities initiated by SUBSCRIBE requests (in either UAS or UAC scenarios) are ended when all subscriptions are terminated. A subscription is terminated when a NOTIFY with a "Subscription-State: terminated" header is responded to (or the response times out). For more information refer to RFC 3265 section 3.3.4.

Dialogs Activities initiated by INVITE requests (in either UAS or UAC scenarios) are terminated by the response to a BYE request (or timed out response to that request).

Dialog Activities initiated by REFER requests (in either UAS or UAC scenarios) are ended when all subscriptions are terminated. A subscription is terminated when a NOTIFY with a "Subscription-State: terminated" header is responded to (or the response times out). For more information refer to RFC 3515.

D.12.4 Mixed SUBSCRIBE, REFER and INVITE scenarios on a single dialog

The behaviour of the RA type for these scenarios is not defined. We recommend that implementers refer to http://www.ietf.org/internet-drafts/draft-ietf-sipping-dialogusage-06.txt or a later version for possible behavior.

D.13 Handling of CANCEL Requests

This RA type simplifies handling of CANCEL requests. The intention of the design is that UAS applications should be able to handle CANCEL scenarios with minimal effort. This is to eliminate the need for each application to have to write “boiler-plate” code when handling CANCEL requests, such as responding to the CANCEL request, and sending a 487 response for the cancelled INVITE server transaction.

D.13.1 RA behaviour when receiving a CANCEL Request

When a CANCEL request is received, the RA will attempt to match the CANCEL with an INVITE server transaction.

If there is a match, the RA fires this cancel event on the INVITE server transaction activity. If the INVITE was a re-INVITE, the cancel event will be fired on the re-INVITE's Dialog activity, since there is no server transaction activity for mid-dialog requests.

If the CANCEL did not match any INVITE, the RA must fire the event on the CANCEL server transaction activity.

The event object for the cancel event is `net.java.slee.resource.sip.CancelRequestEvent`, an extension of `javax.sip.RequestEvent`. CancelRequestEvent contains a reference to the matching INVITE server transaction, if any.

The resource adaptor is expected to check if the CANCEL request was processed by any SBBs. If the CANCEL request was not processed by an SBB, the RA is expected to send a “481 Call or Transaction Does Not Exist”.

The rationale for sending the 481 response here is so that the RA is a good SIP citizen. If the RA knows that no SBBs processed the event, then it can safely assume that there must not be any application state associated with the CANCEL, so it is safe to send an error response and promptly end the CANCEL transaction.

D.13.2 Application behaviour

When the application receives the CancelRequestEvent, it may handle it manually, using the information in the event to generate the appropriate responses as per RFC3261. Alternatively, the application can use the method `SleeSipProvider.acceptCancel()`. This means the RA will automatically generate the appropriate responses.

The method distinguishes between proxy handling and UAS handling as proxies must handle CANCEL differently than a UAS. For more information refer to RFC 3261 § 16.10.

D.14 Handling of Forked Requests

Forked requests occur when a UAC sends a request, and a downstream proxy forwards the request to multiple contacts in parallel. The proxy will forward multiple 1xx and 2xx responses back to the UAC. Each UAS that responds will use a different To-tag in its responses. Each response that arrives at the UAC with a new To-tag creates a new dialog.

D.14.1 UAC Applications

A UAC application in the SLEE must be able to handle the multiple dialogs created in a forking scenario.

When a UAC sends an initial dialog-creating request, the dialog activity has already been created, and is in the Trying state. This will be referred to as the original dialog activity below. The original dialog activity does not yet have a remote tag set, as no responses with To-tags have arrived yet.

When processing responses to the initial dialog-creating request, the RA will check for forked responses, i.e., responses that have a different To-tag because they have come from a different UAS.

When a forked response is detected, the RA shall create a new dialog activity (based on the original dialog, but with a different remote tag), and fire a javax.sip.Dialog.FORKED event. The event is fired on the original dialog activity, so that SBBs attached to the original activity will know that a fork has occurred. The event object (net.java.slee.resource.sip.DialogForkedEvent) contains a reference to the new dialog activity and the response.

SBBs can use this event to detect forks, and attach to the new dialogs so that mid-dialog requests can be sent and received on these new dialogs.

When a 2xx response arrives, the RA finds the dialog that it matches, retains the dialog, transitions it to the Confirmed state, and ends all early dialogs. If the final response was an error response, then the RA ends all dialogs created as a result of the initial request.

D.14.2 Late 2xx responses

Due to proxy forking behavior specified in RFC3261, it is possible for multiple 2xx responses to arrive at the UAC, for the same initial request.

If the application is not using dialogs, the RA shall fire response events for all late 2xx responses, so that a proxy application can forward these responses as required by RFC3261.

If dialog activities are used, only the first 2xx received for the dialog-creating request will be used. Late responses that are retransmits will be ignored, but the RA should retransmit the ACK if necessary. Late responses with new To-tags will also be ignored, but in this case the RA implementation should take appropriate action to tear down the dialog at the UAS, for example send a BYE.

D.14.3 UAC Forking behavior

The following state machine table shows how an RA implementation should process responses to a dialog-creating request. This table is intended to be illustrative of the behavior of the RA when handling UAC dialogs that fork. It is present to specify the behavior an application in the SLEE will observe.

The initial state is AWAIT_FIRST_TAG, and the final state is END. The initial state is entered when the initial request is sent, and the RA is waiting for the first response. If a timeout occurs, this is treated as if a 408 Request Timeout response was received.

Current State	***Received Response***	***Action(s)***	***Next State***
AWAIT_FIRST_TAG	1xx-notag	Fire response event on original	AWAIT_FIRST_TAG

Current State	*Received Response*	*Action(s)*	*Next State*
		dialog.	
AWAIT_FIRST_TAG	1xx-tag	Set remote tag on original dialog. Fire response event on original dialog.	AWAIT_FINAL
AWAIT_FIRST_TAG	2xx-tag	Set remote tag on original dialog. Fire response event on original dialog.	END
AWAIT_FIRST_TAG	3xx-6xx	Fire response event on original dialog. End original dialog.	END
AWAIT_FINAL	1xx-notag	Fire response event on original dialog.	AWAIT_FINAL
AWAIT_FINAL	1xx-tag (tag matches existing early dialog)	Fire response event on matching dialog.	AWAIT_FINAL
AWAIT_FINAL	1xx-tag (tag does not match an existing early dialog)	Create new dialog activity in Early state. Fire Dialog.FORKED event on original dialog.	AWAIT_FINAL
AWAIT_FINAL	2xx-tag (tag matches existing early dialog)	Transition matching dialog to Confirmed state. End other early dialogs. Fire response event on matching dialog.	END
AWAIT_FINAL	2xx-tag (tag does not match an existing early dialog)	Create new dialog activity in Confirmed state. Fire Dialog.FORKED event on original dialog. End other early dialogs.	END
AWAIT_FINAL	3xx-6xx	Fire response event on original dialog. End all early dialogs.	END

D.14.3.1.1 B2BUA and Forking behavior

B2BUA applications may need to forward forked responses upstream (for example if they are transparent B2BUAs). These responses can create dialogs at the UAC, therefore such a B2BUA needs to create a corresponding UAS dialog activity, in order to receive any mid-dialog requests on these dialogs.

A B2BUA that is forwarding forked responses should use the `SleeSipProvider.forwardForkedResponse()` method to send forked responses upstream. This method returns a new UAS dialog activity, which the B2BUA should attach to, so that it receives mid-dialog requests.

As with UAC dialogs, the first final response that is sent upstream will end any non-matching early dialogs, so that one confirmed dialog remains.

Appendix E JAIN TCAP Resource Adaptor Type

This appendix describes a *recommended* adaptation of a JAIN TCAP resource to the SLEE. JAIN TCAP is a Java representation of the Transaction Capabilities Application Part layer of a Signaling Systems 7 (SS7) protocol stack. The JAIN TCAP specification can map to the ITU 1993, ITU 1997, ANSI 1992 and ANSI 1996 versions of the TCAP protocol.

E.1 Resource adaptor type identifier

The resource adaptor type name is "JAIN TCAP". The resource adaptor type vendor is "jain.protocol.ss7.tcap". The resource adaptor type version is "1.1".

E.2 Activity objects

The Activity objects of a JAIN TCAP resource are `Java.lang.Integer` objects that encapsulate integer `dialogueId`s. Allocation of dialogueId's in the TCAP stack is often the responsibility of the TCAP stack, however JAIN TCAP provides the ability to request a new dialogueId from the TCAP stack with the `SipProvider.getNewDialogueId` method, which in turn would map to a new Activity in the resource adapter.

Sending an `AbortReqEvent` and `EndReqEvent` message on the `SipProvider` terminate the local `Activity` object, i.e. `AbortIndEvent` and `EndIndEvent` terminate the remote `Activity` object, the resource adapter must terminate its local representation of the `Activity` object upon the local users decision to send the Request messages.

E.3 Events

The following table lists the events emitted by a JAIN TCAP resource and its Activity objects.

Events	***java.lang.Integer containing dialogueId***	***Remarks***
DIALOGUE EVENTS		
BeginIndEvent	X	1.
ContinueIndEvent	X	
EndIndEvent	X	2.
NoticeIndEvent	X	
ProviderAbortIndEvent	X	3.
UnidirectionalIndEvent	X	4.
UserAbortIndEvent	X	5.
COMPONENT EVENTS		
InvokeIndEvent	X	
ResultIndEvent	X	
ErrorIndEvent	X	
RejectIndEvent	X	
LocalCancelIndEvent	X	

1. This event represents a new dialogue in the underlying JAIN TCAP resource (or Activity object in the JAIN TCAP resource adaptor entity) initiated by the remote user, which is represented as a new Activity Context within the SLEE.

2. This event signals that there are no more events on the dialogue identified by the `dialogueId` attribute of the event object. An `EndReqEvent` event alerts the JAIN TCAP resource that the dialogue (or Activity object) has ended. The resource adaptor entity that represents the JAIN TCAP resource should signal to the SLEE that the Activity object has ended on receipt of an `EndIndEvent` event.

3. This event signals that the underlying JAIN TCAP resource has abruptly ended the specified dialogue. The dialogue has been terminated by the `JainTcapProvider` in reaction to a transaction abort by the transaction sub-layer. Any pending components on the Activity object are not transmitted. The resource adaptor entity that represents the JAIN TCAP resource should signal to the SLEE that the Activity object has ended on receipt of a `ProviderAbortIndEvent` event.

4. The `UnidirectionalIndEvent` event is used by the JAIN TCAP resource as a means to pass components to an SBB when no reply is expected. This primitive is used when there is no need to establish a dialogue with another TC User, however an Activity object should be created in order to route the primitives in the SLEE.

5. This event signals that the remote TCAP user has abruptly ended the specified dialogue. The resource adaptor entity that represents the JAIN TCAP resource should signal to the SLEE that the Activity object has ended on receipt of an `AbortIndEvent` event.

E.3.1 Event types

The event type name of an event type is the fully qualified name of the class for the event type. For example, the event type name of `BeginIndEvent` is "jain.protocol.ss7.tcap.dialogue.BeginIndEvent".

The event type vendor and event type version are "jain.protocol.ss7.tcap" and "1.1", respectively.

E.3.2 Event classes

The event classes for each of these event types are defined by JAIN TCAP.

- The event classes of dialogue primitives all extend the `jain.protocol.ss7.tcap.DialogueIndEvent` abstract class.
- The event classes of component primitives all extend the `jain.protocol.ss7.tcap.ComponentIndEvent` abstract class.

E.4 Activity Context Interface Factory interface

The interface of the JAIN TCAP resource adaptor type specific Activity Context Interface Factory should be as follows:

```
…
import javax.slee.ActivityContextInterface;
import javax.slee.FactoryException;
import javax.slee.UnrecognizedActivityException;
…
public interface JainTcapActivityContextInterfaceFactory {
       public ActivityContextInterface
              getActivityContextInterface(Integer dialogueId)
                     throws  NullPointerException,
                             UnrecognizedActivityException,
                             FactoryException;
}
```

E.5 Resource adaptor object

The resource adaptor object that is bound to an SBB's component environment when the SBB uses a `resource-adaptor-entity-binding` deployment descriptor element to reference a JAIN TCAP resource is a `JainTcapProvider` object, i.e. the `resource-adaptor-interface-name` element contains `jain.protocol.ss7.tcap.JainTcapProvider`.

E.6 Restrictions

The following restrictions apply:

- Only the SLEE can add itself as a listener. The JAIN TCAP resource or its resource adaptor should not allow any other objects to add themselves as listeners. An attempt to do so may be rejected by throwing a `SecurityException`.

Appendix F Integration with EJB applications

This appendix specifies how SBBs may invoke EJBs and how EJBs may pass events to the SLEE.

F.1 Invoking an EJB

From the EJB specification perspective, an SBB is a remote client (see Sections 6.1 and 9.2 of the "Enterprise JavaBeans 2.0, Final Release" specification).

An SBB may invoke an EJB through the use of the EJB's home interface. The home interface reference is gained through the SBB's component environment.

The location of a home interface in the SBB's component environment and the type of the home interface are specified using the `ejb-ref` element in the SBB's deployment descriptor. For more information see Section 3.1.8.

EJB components have transactional semantics. For more information regarding possible transaction scenarios, see Section 9.9.

F.1.1 Example code

This following example shows how an SBB may obtain a reference to the home interface of an EJB component.

```
        …
        // obtain the SBB component environment naming context.
        Context initCtx = new InitialContext();
        Context myEnv = (Context)initCtx.lookup("java:comp/env");
        …
        // get reference to the home object
        Object homeRef = myEnv.lookup("location/of/home/interface");
        ParticularEjbHome home = (ParticularEjbHome)
                PortableRemoteObject.narrow(homeRef, ParticularEjbHome.class);
        …
        // invoke methods on the home object
        try{
                home.remove(somePrimaryKeyObject);
        } catch ( … )
                …
        }
```

F.2 Passing an event to SLEE

An EJB component may pass an event to the SLEE in order to drive application logic contained within the SLEE. This section describes the application visible API that can be used to send events to the SLEE. An EJB developer is expected to understand the EJB model for using external resources in order to use this API. For more information see the EJB 2.0 specification and the JCA 1.0 specification.

The EJB developer may use the following APIs to pass events to SLEE.

```
package javax.slee.connection;

public interface ExternalActivityHandle extends java.io.Serializable {
        public boolean equals(Object other);
        public int hashCode();
}

public interface SleeConnection{
        public ExternalActivityHandle createActivityHandle();
        public void fireEvent(Object event, javax.slee.EventTypeID eventType,
                              ExternalActivityHandle activityHandle,
                              javax.slee.Address address);
        public javax.slee.EventTyeID getEventTypeID(String name, String vendor,
                              String version);
        public void close();
```

```
}

public interface SleeConnectionFactory {
        public SleeConnection getConnection();
}
```

The connection factory instance gained by an EJB component will implement the `javax.slee.connection.SleeConnectionFactory` interface. This interface allows EJB components to obtain `javax.slee.connection.SleeConnection` instances representing connections to an external SLEE as needed via the `SleeConnectionFactory.getConnection` method.

A `SleeConnection` instance allows an EJB component to create external activity handles, retrieve SLEE event type identifiers, and fire events to the SLEE. Each `SleeConnection` remains open until explicitly closed via the `SleeConnection.close` method.

To ensure that the SLEE can correctly identify and deserialize events sent via the `SleeConnection` interface, each event fired has an associated event type obtained via the `SleeConnection.getEventTypeID` method that identifies the event type by name, vendor, and version. The class of the event type specified should be compatible with the actual type of the event object fired.

Events fired via the `SleeConnection` interface are delivered to Null Activities. The creation of these activities is managed by the SLEE in response to external event firing, and they are subject to the normal null activity garbage-collection rules.

EJB components create one or more external activity handles via the `SleeConnection.createActivityHandle` method. The `ExternalActivityHandle` interface represents these handles. External activity handles can be used to fire events via the `SleeConnection` that created them; the visibility of the created handles to other `SleeConnection` instances is implementation-defined.

The mapping between external activity handles and null activities is implementation-specific, but must have the following characteristics:

- Firing an event with an external activity handle that is not mapped to an existing null activity will cause the creation of a new null activity associated with the external activity handle.
- Subsequent events with the same handle are delivered to the same activity, while the activity exists.
- When a null activity ends, any association with an external activity handle is cleared. If a subsequent event is fired using this handle, it is treated as though there is no mapping (i.e. a new activity will be created).

The `SleeConnection.createActivityHandle`, `SleeConnection.getEventTypeID`, and `SleeConnection.close` methods do not have transactional behavior.

The `SleeConnection.fireEvent` method has required transactional behavior. If a transaction is in progress when this method is called, processing of the fired event is deferred until this transaction is known to have committed. Failure to fire the event is not required to cause rollback of the enclosing transaction. Within a transaction, delivery order of fired events is preserved.

A successful commit of a transaction that fires events does not guarantee that the fired events will be processed by the SLEE. EJB and SBB developers should accommodate the possibility of event loss (for example, via timers).

A SLEE implementation is not required to provide an implementation of these interfaces.

F.2.1 Example code

The following example shows how an EJB component may fire events to an external SLEE.

```
// Obtain a reference to the connection factory from JNDI
InitialContext ctx = new InitialContext();
SleeConnectionFactory factory =
```

```
  (SleeConnectionFactory)ctx.lookup("java:comp/env/slee/MySleeConnectionFactory");

// Obtain a connection to the SLEE from the factory
SleeConnection connection = factory.getConnection();

try {
  // Locate the event types for two different events
  EventTypeID eventTypeOne = connection.getEventTypeID("MyEventOne","MyVendor","0.1");
  EventTypeID eventTypeTwo = connection.getEventTypeID("MyEventTwo","MyVendor","0.1");

  MyEventOne eventOne = ...;
  MyEventTwo eventTwo = ...;

  // Fire two events on a newly created null activity
  ExternalActivityHandle handle = connection.createActivityHandle();
  connection.fireEvent(eventOne, eventTypeOne, handle, null);
  connection.fireEvent(eventTwo, eventTypeTwo, handle, null);
} finally {
  // Done with the connection. Close it.
  connection.close();
}
```

Appendix G Frequently Asked Questions

G.1 Persistence

The SLEE specification uses the term "persistent" to describe state that can survive certain documented failures. Examples of persistent state include CMP fields of SBB entities, Activity Contexts (attached SBB entities and attributes), SLEE defined Activity objects, Profile CMP fields, and Timers.

However, the SLEE does not define what "persistence" means. This is intentional to allow SLEE vendors to provide different levels of persistence. Individual SLEEs can be optimized or customized to the requirements of their applications or operational environments. In some cases, a single SLEE may provide different persistence levels to support different application or operational requirements.

In general, a better or higher persistence level can maintain persistent state across more types of failures but has poorer performance because of higher processing overhead and higher transactional latencies.

For example, one level of persistence may store persistent state in disk based stable storage. Persistent state is maintained across complete failures of the SLEE, e.g. all nodes of the SLEE cluster fails. This level of persistence is typical in commercial transaction processing systems. However, it may not be appropriate for real-time communications applications because of relatively long disk access latencies to commit persistent state to stable storage.

Another level of persistence is to replicate persistent state to the main memory of another node in a multi-node SLEE cluster. Persistent state may be lost if certain state replicas fail or if there is a total power failure. This level of persistence is typical in real-time communications applications because the latency to replicate state to the memory of another node can be significantly less than storing persistent state to disk-based stable storage.

SLEE vendors are expected to document the persistence capabilities provided by their SLEEs. For example, the SLEE vendor should document whether the SLEE maintains persistent state across graceful administrative shutdown and restart of the SLEE.

G.2 SLEE event model versus JavaBeans listener model

Event consumers in the JavaBeans event model subscribe for events directly with event producers via add listener methods. In this model, an event producer has direct knowledge of its event consumers and is responsible for delivering events to the subscribed consumers.

The SLEE event model is based on the publish/subscribe model (similar to JMS). Event consumers subscribe for events by attaching to Activity Contexts. Event producers publish events to Activity Contexts. An event producer is not directly aware of its event consumers and an event consumer is not directly aware of its event producers. The SLEE implemented Activity Contexts maintain the relationships among event producers and event consumers.

The publish/subscribe model was chosen by the SLEE expert group because it has the following benefits:

- It allows the SLEE to know the relationships among event consumers and event producers. This allows the SLEE to provide important value added features like garbage collection of event consumers that will no longer receive events.
- It eliminates direct references among event producers and event consumers. Direct references reduce application robustness because they introduce the possibility of invalid or dangling references.
- It allows the SLEE to specify a single consistent event distribution model (e.g. the order of event delivery to multiple event consumers, concurrency control semantics). A single model benefits the developer since the developer only has a single model to learn. It also improves robustness because event producers do not have to implement event distribution code that complies with documented patterns or conventions.

- It promotes stronger decoupling or isolation of the event producer from the event consumer. This facilitates modularity and fault isolation, i.e. an error in the event producer is less likely to propagate to the event consumer and vice versa.

While the JavaBeans listener model has certain advantages, the SLEE expert group felt that the publish/subscribe model allows the SLEE to provide more value-add to event oriented applications.

G.3 SLEE originated invocation vs. non-SLEE originated invocation design

When application code calls application code it is common for the caller to know about the set of exceptions that the callee can throw. In this case the caller is responsible for performing the appropriate action based on the exception that was thrown.

This is the case for `SbbLocalObject` invocations, i.e. when one SBB invokes another SBB through an `SbbLocalObject` contract.

When the SLEE invokes an SBB, there is no caller that knows how to perform the appropriate action if the exception was thrown. For example the SLEE does not know how to play an announcement on a telephone call (assuming a telephone call is the activity). Such error handling logic is application specific. Therefore when the SLEE calls an SBB and something goes wrong, the SLEE delegates the problem back to the SBB. This could be viewed as a circular design, but it allows the SBB developer to put their application logic in one place (the `sbbExceptionThrown` method) and if the SBB developer does not like such an approach they are free to place all event handling code in a `try { ... } catch (Throwable t) { .. }` statement.

SLEE always delivers events to SBBs in the context of a transaction. It is possible that the transaction rolls back without the SBB knowing that the transaction will rollback when it is handling an event. For example the SBB event handler method returns, and at is point of return the `getRolledBackOnly()` returns `false`. As the SLEE attempts to commit the transaction the transaction manager refuses to commit the transaction, because a resource involved in the transaction indicates some sort of error. In this case the environment had a problem that the application is unaware of. Typically in call control applications transactions are used to demarcate points in the applications state machine within a well understood failure model, and the application is written such that if a transaction does not commit, that the application stops taking an active roll in the call. In order to support the application stop taking an active roll in the call, the SBB developer can detach from an activity or perform some other transacted operation in its `sbbRolledBack` callback.

G.4 Writing an `sbbRolledBack` method

The SBB developer should write their rolled back handler methods to cope with a transaction that contains an event handler only, a remove only, or an event handler and remove rolling back. This can be done by using the following approach:

```
public void sbbRolledBack(RolledBackContext ctx){
```

```
    if ( ctx.getEvent() != null ){
          // do any event handling cleanup necessary
    }

    if( ctx.isRemoveRolledBack() ) {
       // do any cleanup necessary to handle a remove that didnt commit
    }

}
```

It should be noted that whether or not any cleanup needs to be performed if a transaction committed or not is application specific.

G.5 Profiles and Java Persistence technologies

Profiles support the minimum needs of communications applications in a container environment, for example number translation tables, prefix based searches on address strings etc. It is designed explicitly to enable high throughput and low latency access, and is made possible through a simple API that can be implemented in a very performant manner.

It is not intended to be a sophisticated persistence model that supports rich and potentially complex data models. There are many frameworks available for the Java platform for such uses. The user should be aware of the performance characteristics of their chosen persistence framework, implementation characteristics etc, such that it meets their needs.

The SLEE's Profile API enables the SLEE vendor to provide implementations that meet performance representations.

The interfaces, while simple, are sufficient to satisfy a large proportion of the needs of communications applications. Most communications applications do not require rich data models.

Glossary

Activity	An Activity object represents a related stream of events. These events represent occurrences of significance that have occurred on the entity represented by the Activity. From a resource's perspective, an Activity represents an entity within the resource that emits events on state changes within the entity or resource.
Activity Context	An Activity Context represents and encapsulates an Activity object in the SLEE domain. An Activity Context is also a store for attributes that may be shared by multiple SBB entities that interact with the Activity object represented by the Activity Context. An Activity Context is also a channel that both emits events, and accepts events fired on the Activity Context.
Activity Context attribute	An Activity Context attribute is an attribute that an SBB component defines in its SBB Activity Context Interface interface as a set of get and set accessor methods. These accessor methods provide a type-safe mechanism for accessing shareable attributes stored in Activity Contexts.
Activity Context Interface object	An Activity Context Interface object is a Java object that implements the SLEE specification defined `ActivityContextInterface` interface and may optionally implement an extended SBB Developer defined SBB Activity Context Interface interface. It provides an Activity Context with a Java object that SBB objects can invoke. An Activity Context Interface object can be either a generic Activity Context Interface object or an SBB Activity Context Interface object.
Activity object	An Activity object is a Java object that provides an Activity with a visible Java API that can be invoked. Activity objects are defined, created and owned by resources, resource adaptor entities or SLEE Facilities. The `getActivity` method defined in the generic Activity Context Interface interface returns an Activity object when invoked. Furthermore, an Activity object may be passed to an Activity Context Interface Factory object to obtain a generic Activity Context Interface object that represents the Activity Context of the Activity object.
`ActivityContextInterface` interface	An `ActivityContextInterface` interface is a Java interface that is defined by the SLEE specification. It defines the operations that all SBB objects can perform on an Activity Context. It does not declare any shareable Activity Context attributes.
address	An address is a unique identifier for an entity in a network. A common address today is a telephone number or an IP address.
Address Profile	An Address Profile is a Profile that has at least an `addresses` attribute that holds an array of `Address` objects.
Address Profile Table	An Address Profile Table contains Address Profiles and is created from an Address Profile Specification. The Address Profile Table of a Service contains provisioned addresses that may cause new root SBB entities of the Service to be instantiated.

Address Profile Specification	An Address Profile Specification specifies the attributes of the Address Profile Tables created from the Address Profile Specification. One of these attributes must be an `addresses` attribute. The Java type of this attribute must be an `Address` object array.
`Address` object	An `Address` object represents an address and is an instance of the `Address` class.
Administrator	An Administrator manages the SLEE and the services running in the SLEE through the SLEE's management interfaces. An Administrator may be a management client, such as a network management application, network management console, or a person using a management client to manage the SLEE.
attachment count	The attachment count of a parent SBB entity is the sum of the attachment counts of all its child SBB entities and the number of Activity Contexts that the parent SBB entity is directly attached to.
child relation (in the context of SBBs)	A child relation relates a child SBB to a parent SBB. It also specifies the default event delivery priority for the child SBB entities created on this child relation.
child relation (in the context of SBB entities)	A child relation relates a child SBB entity to a parent SBB entity. It also specifies the event delivery priority of child SBB entity.
custom event type	A custom event type is an event type introduced by an SBB Developer.
default event delivery priority	The default event delivery priority of a child relation specifies the initial event delivery priority of an SBB entity created on the child relation.
default Profile	The default Profile of a Profile Table provides the initial values for the attributes of new Profiles of the same Profile Table.
deployable unit	A deployable unit is a jar file that can be installed in the SLEE. A deployable unit may contain Services, SBB jar files, event jar files, Profile Specification jar files, resource adaptor type jar files, and resource adaptor jar files. Each of these jar files contain the Java class files and the deployment descriptors of one or more of these components.
event	An event represents an occurrence of significance that has occurred. An event is represented in the SLEE by an event type and an event object.
event class	The event class of an event object is the publicly declared Java class or interface of the event object. The event class may not be the concrete implementation class of the event object.
event delivery priority	The event delivery priority of an SBB entity determines when the SBB entity will receive an event relative to the other SBB entities that can also receive the event.
event name	An event name is an SBB scoped local name that refers to an event type. This name must conform to the naming requirements for Java method names. The event name of the event type is specified by the `event-name` element of the `event` element in the SBB's deployment descriptor.
event object	An event object is a Java object that represents the event and carries information to be conveyed. It cannot be a primitive Java type.

event type	The event type of an event completely determines how the SLEE will route the event, i.e. which SBB entities will receive the event, and which event handler methods will be invoked to process the event. All events with the same event type are represented by event objects of the same event class.
generic Activity Context Interface object	A generic Activity Context Interface object is a Java object that implements the generic `ActivityContextInterface` interface but does not implement an extended SBB Developer defined SBB Activity Context Interface interface. It can only be used to access the underlying Activity object of an Activity Context, attach and detach SBB entities to the underlying Activity Context.
initial event	An initial event is the event that caused the SLEE to instantiate a new root SBB entity. This initial event is known as the initial event of the SBB entity.
management client	A management client is a program that manages the SLEE and the applications running in the SLEE through the SLEE's management interfaces. For example, a management client may be a network management application or a network management console.
Object Name	An Object Name is the name used to identify a specific JMX MBean object registered with a JMX MBean Server.
Profile	A Profile contains a set of attributes that hold data provisioned by the Administrator at runtime via a management interface and is accessible by SBB objects running in the SLEE. It exists in a Profile Table. The Profile Specification of its Profile Table specifies its attributes and the management interface used to provision the Profile.
Profile CMP interface	A Profile CMP interface defines the attributes of each Profile in the Profile Tables created from the Profile Specification that defines the Profile CMP interface. The SBB Developer declares each attribute using a pair of get and set accessor methods in this interface.
Profile Management abstract class	A Profile Management abstract class is a Java class that implements the Profile Management interface of the Profile Specification. It provides the implementation of the methods declared in the SLEE specification defined `ProfileManagement` interface.
Profile Management interface	A Profile Management interface is a Java interface that defines the management operations (and may include the CMP field get and set accessor methods declared in the Profile CMP interface) that interact and manipulate the attributes of a Profile that conforms to the Profile Specification that defines the Profile Management interface and should be exposed through the management interface of the Profile Specification.
Profile Management object	A Profile Management object is either a Java object that is an instance of a concrete class that extends the Profile Management abstract class if one is provided, or a Java object that is an instance of a concrete class that implements the Profile CMP interface and the `javax.slee.profile.ProfileManagement` interface. It caches the persistent state of a Profile in its CMP fields and instance variables.
Profile MBean class	A Profile MBean class is a SLEE implemented class that implements the corresponding Profile MBean interface.

Profile MBean interface	A Profile MBean interface is the external management interface that is visible to external management clients that interact with the Profiles specified by the Profile Specification. It is generated by the SLEE upon Profile Specification deployment from the methods defined in the Profile CMP interface if no Profile Management interface is provided or the Profile Management interface if the Profile Management interface is provided, and the SLEE specification defined `javax.slee.profile.ProfileMBean` interface.
Profile MBean object	A Profile MBean object is an instance of a Profile MBean class.
Profile Specification	A Profile Specification defines the interfaces, classes, and deployment descriptor elements needed to define the attributes of Profiles in Profile Tables created from the Profile Specification and the interfaces used to provision and access a Profile of the Profile Specification.
`ProfileManagement` interface	The `ProfileManagement` interface is a SLEE specification defined interface that declares the life cycle methods of a Profile Management object.
`ProfileMBean` interface	The `ProfileMBean` interface is a SLEE specification defined interface that declares the life cycle methods of a Profile MBean object.
resource adaptor	A resource adaptor is an implementation of a resource adaptor type. It mainly consists of Java classes that implements the function of adapting a resource to the requirements of the SLEE. An Administrator installs resource adaptors in the SLEE.
resource adaptor entity	A resource adaptor entity is an instance of a resource adaptor. Typically, a resource adaptor. An Administrator instantiates resource adaptor entities from resource adaptors.
resource adaptor type	A resource adaptor type declares the common characteristics of a set of resource adaptors. For example, it defines the Activity objects, the event types, the resource adaptor interface, and the Activity Context Interface Factory interface that are common to these resource adaptors.
Resource Info Profile Table	The Resource Info Profile Table of a Service contains Service specific information that should be passed to resource adaptors when the Service is in the Active state.
root SBB	A root SBB is an SBB that may be instantiated by the SLEE to process events.
root SBB entity	A root SBB entity is the root node in an SBB entity tree and is an instance of a root SBB. It is instantiated by the SLEE to process its initial event.
SBB	Same as SBB component.
SBB abstract class	An SBB abstract class is a part of an SBB component. It contains the core application logic of the SBB component, including how events should be processed and the implementations of the methods declared in its SBB local interface. It also defines the CMP fields of the SBB abstract class.

SBB Activity Context Interface interface	An SBB Activity Context Interface interface is a part of an SBB component. It extends the generic SLEE specification defined `ActivityContextInterface` interface and defines a set of get and set accessor methods for each Activity Context attribute accessed by the SBB component.
SBB Activity Context Interface object	An SBB Activity Context Interface object is a Java object that implements the SBB Activity Context Interface interface of an SBB component. It provides an Activity Context with a Java object that SBB objects can use to access the Activity Context attributes that are defined in the SBB Activity Context Interface interface of the SBB component and stored in the Activity Context.
SBB component	An SBB component consists of a set of Java classes and interfaces, and a deployment descriptor that describes the SBB component. These classes and interfaces contain the SBB component's application code.
SBB component identifier	An SBB component identifier is a Java object that uniquely identifies an SBB component.
SBB Developer	An SBB Developer is a person who creates SBB components.
SBB entity	An SBB entity is an instance of an SBB component. It represents the persistent per-instance state of an instance of the SBB component as defined by the CMP fields in the SBB abstract class of the SBB component.
SBB entity sub-tree.	An SBB entity sub-tree is an SBB entity tree whose root is not the root SBB entity of the SBB entity tree.
SBB entity tree	An SBB entity tree is a directed acyclic graph that represents the child relations between SBB entities. A node in this tree represents an SBB entity. A directed edge represents the child relation from a parent SBB entity to a child SBB entity. The label on the directed edge indicates the event delivery priority of the child SBB entity relative to the child's siblings. The root node of this tree must be a root SBB entity.
SBB graph	An SBB graph is a graph that illustrates the child relations among SBBs. An SBB is a node in this graph and a child relation is a directed edge from the node of the parent SBB to the node of the child SBB. Each edge has a label that indicates the default event delivery priority of the child relation.
SBB local interface	An SBB local interface declares the methods of the SBB component that may be invoked synchronously. The SBB Developer provides the implementation of these methods in the SBB abstract class.
SBB local object	An SBB local object is an instance of a SLEE generated concrete class that implements the SBB local interface of an SBB. This SBB local object represents an SBB entity. When invoked, it delegates the invocation to an SBB object assigned to the SBB entity.
SBB object	An SBB object is an instance of an SBB abstract class. It has a life cycle. When an SBB object is in the Ready state, the SBB object has been assigned to an SBB entity. It executes on behalf of the SBB entity and it can access the persistent state of the SBB entity.

`SbbContext` object	The `SbbContext` object of an SBB object gives the SBB object access to the SBB object's context maintained by the SLEE, allows the SBB object to invoke functions provided by the SLEE. It also allows the SBB object to obtain runtime information about the SBB entity assigned to the SBB object.
Service	Same as Service component.
Service component	A Service component specifies a child relation from the SLEE (as the logical parent of all root SBBs) to a root SBB, the default event delivery priority of the Service, and provides other information needed by the SLEE to instantiate root SBB entities of the Service, such as the Service's Address Profile Table and Resource Info Profile Table.
Service Deployer	A Service Deployer is a person who customizes SBBs and Services for deployment into the target operational environment.
Service component identifier	A Service component identifier is a Java object that uniquely identifies a Service component.
subscriber	A subscriber is an entity; it may be a person, business etc.