ANDERSEN
CONSULTING

Eagle Architecture Specification

EAGLE ARCHITECTURE MATERIALS LICENSE AGREEMENT

Andersen Consulting (AC) grants to you a non-exclusive license to use the accompanying materials provided you agree to the following terms and conditions: You may use the materials for your own personal use and for your internal business use as an owner or employee of your company. You may copy the materials provided they are subject to this license agreement and that all copies contain all of the original copyright and proprietary notices. You may not rent, lease, or let others use the materials and you may not modify, translate, reverse engineer, decompile, disassemble, or create derivative works based on the materials. This license agreement and the rights to use the materials cannot be assigned, sublicensed, or transferred and any attempt to do so is void. Title and all intellectual property rights in the materials and to all copies of the materials and derivatives of the materials will be owned by AC or its suppliers. You are solely responsible for compliance will all applicable laws, including without limitation, all applicable export laws, rules and regulations. To the extent permitted under applicable law, AC provides the materials "as is" and with all faults, and AC does not make any warranty of any kind with respect to the materials. In all other cases, AC’s limited warranty is that, for a period of ninety days, the materials are the most recent version of the materials that AC has developed and your sole and exclusive remedy is to have AC replace any nonconforming materials with the most recent version of the materials.

AC MAKES NO OTHER WARRANTIES RELATED TO THE MATERIALS, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN EXCHANGE FOR YOUR RECEIPT OF THE MATERIALS FOR NO FEE, YOU AGREE THAT AC WILL NOT BE LIABLE TO YOU OR ANY OTHER PERSON FOR ANY DAMAGES OF ANY KIND, AND THAT AC WILL NOT BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF AC KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION OF LIABILITY SHALL NOT APPLY IF APPLICABLE LAW PROHIBITS SUCH LIMITATION. FURTHERMORE, SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS LIMITATION AND EXCLUSION MAY NOT APPLY TO YOU.

This license agreement represents the complete agreement of the parties related to the materials and supersedes all prior agreements and representations between us and may only be amended by a writing executed by both parties. If any provision of this license agreement is held to be unenforceable for any reason, such provision shall be reformed

Introduction

Overview

Eagle has focused on providing an architecture that makes technology support our clients' business strategies and enforces a robust, predictable, and consistent application model. This architecture provides the foundation for building systems that are agile, architected for business change, flexible, and better able to meet the needs of a client's business.

The Eagle Architecture Specification (EAS) defines the architecture in a language-independent form. The EAS is a proven blueprint Andersen Consulting uses to help clients in the banking, insurance, telecommunications, and manufacturing industries build component-based, enterprise solutions. These solutions are:

- Powerful
- Flexible
- Leveraged by object technology and component reuse
- Scaleable
- Distributable

The specification represents a collection of frameworks and design patterns for integrating and building components.

For enterprise application developers, the EAS provides a standard for structuring applications to enable business component sharing and reuse.

For software tool developers, the EAS offers a benchmark for the types of tools that are needed to construct highly flexible, open, component-based business applications.

About This Document

This document contains the specifications for all of the frameworks that make up the Eagle Architecture Specification. This document provides tool and application architects with abstract framework designs that they can customize and implement to develop scaleable, component-based, business solutions.

About Frameworks

"Projects fail despite the latest technology for lack of ordinary solutions"

- founders of Pattern Languages of Programming (PLoP)

The value offered by component and object technology is not inherent in the technology itself, but rather lies in the quality of how the technology is applied. The old adage, "it's how you use it" applies. The terms *frameworks* and *patterns* have become popular ways of describing how technology should be applied to a specific problem. Over the years, the terms *architecture* and *application architecture* have also been used to describe this level of an application design.

While it is clear that quality frameworks and patterns are essential to building large, industrial-strength solutions, several challenges are associated with their availability:

- Frameworks and patterns for distributed, componentized, and / or object oriented applications are scarce, and the old patterns don't work. Developers have had to invent frameworks and patterns concurrent with the building of the application in question.
- Framework and pattern developers are scarce; relatively few people have experience to span both framework and application development.
- Development projects are reluctant to divert energy from the short-term goal of deploying functionality towards the longer-term promise of payback through improvement of systems-building constructs, thus improving the qualities of the resulting applications.
- Some organizations view this level of knowledge as just part of what makes a good developer and attempt to hire people "that already know this."
- Some developers view an institutionalization of frameworks and patterns as a practice that takes the art out of systems building; they are not interested in supporting or investing in types of things that might constrain their ability to code from scratch.
- Development projects cannot overcome the "chicken and egg" problem: building the framework capabilities in time to meet the application requirements and identifying the application requirements soon enough to allow time for developing the framework capabilities.

To try and help the industry overcome these hurdles and to provide some basis for standardization, Eagle has made available frameworks and patterns for building enterprise solutions from distributed components.

About the Eagle Architecture Specification

The EAS represents a set of the frameworks and design patterns that Andersen Consulting uses to construct scaleable, distributable, component-based, enterprise solutions. The frameworks contained in the EAS are:

- Security
- Core Component Integration
- Workflow Facility
- Component Interface
- Activity Control
- Business Class Modeling / Persistence
- User Interface Control Framework
- Integrated Performance Support (IPS)

The benefits achieved by using these frameworks include:

- Flexible, scaleable enterprise solutions assembled from reusable components.
- Flexible components built with a proven, object-oriented application architecture.

In addition, developers can use the EAS frameworks in collaboration with Andersen Consulting's OO processes and methodologies, which support the analysis, design, and construction of component-based enterprise solutions. This collaboration results in a seamless transition from analysis and design to implementation.

The specification is:

- Open. It's language-independent and intended for sharing among clients, vendors, and standards bodies (e.g., OMG).
- Flexible. It's not an implementation; you can interpret and implement the specification according to special circumstances.
- Extensible. You can add and extend frameworks to the specification.
- Accessible. It's published via the Web.

Problems that the Specification Attempts to Address

Developers typically encounter a "lack of partitioning" problem when modifying or extending an object-based, enterprise solution. Without partitioning:

- A change made to a single application feature can ripple through the entire solution. The solution may fail to meet performance and scalability requirements due to inadequate distribution and configuration flexibility.

The EAS addresses the lack of partitioning problem by encouraging developers to apply the Separation of Concern pattern and several component-specific frameworks to their applications. At the enterprise level, EAS includes frameworks designed to help the developer define and integrate components. These frameworks introduce partitioning and separation of concern at the enterprise-solution level. At the component level, the EAS includes frameworks designed to help the developer build and introduce partitioning within a component.

Assumptions, Notation, and Presentation

We've written the EAS in the context of two specifications:

- CORBA
- Unified Modeling Language Metamodel

We chose these two specifications for their openness, acceptance by the solutions-building community, and their applicability to the EAS goals. However, you can express the EAS in the context of other specifications, such as Active X/COM, and we expect that the solutions-building community will do so as a natural by-product of the work we accomplish with our clients.

To provide a consistent, formal syntax for expressing the EAS, we've specified the EAS frameworks using Java and IDL syntax. (We used Java to express the APIs of the Building Components Frameworks and IDL to express the APIs of the Integrating Components Frameworks.) However, you can translate the EAS frameworks into other syntaxes, such as C/C++. Again, we expect the solutions-building community will do so as a result of our work with clients.

Frameworks Overview

Introduction

We've organized the EAS frameworks into categories and subcategories for clarity and presentation purposes; this categorization does not constrain collaboration among the frameworks. Objects collaborate across the frameworks, as necessary, in order to accomplish specific, enterprise solution objectives.

The two frameworks categories are:

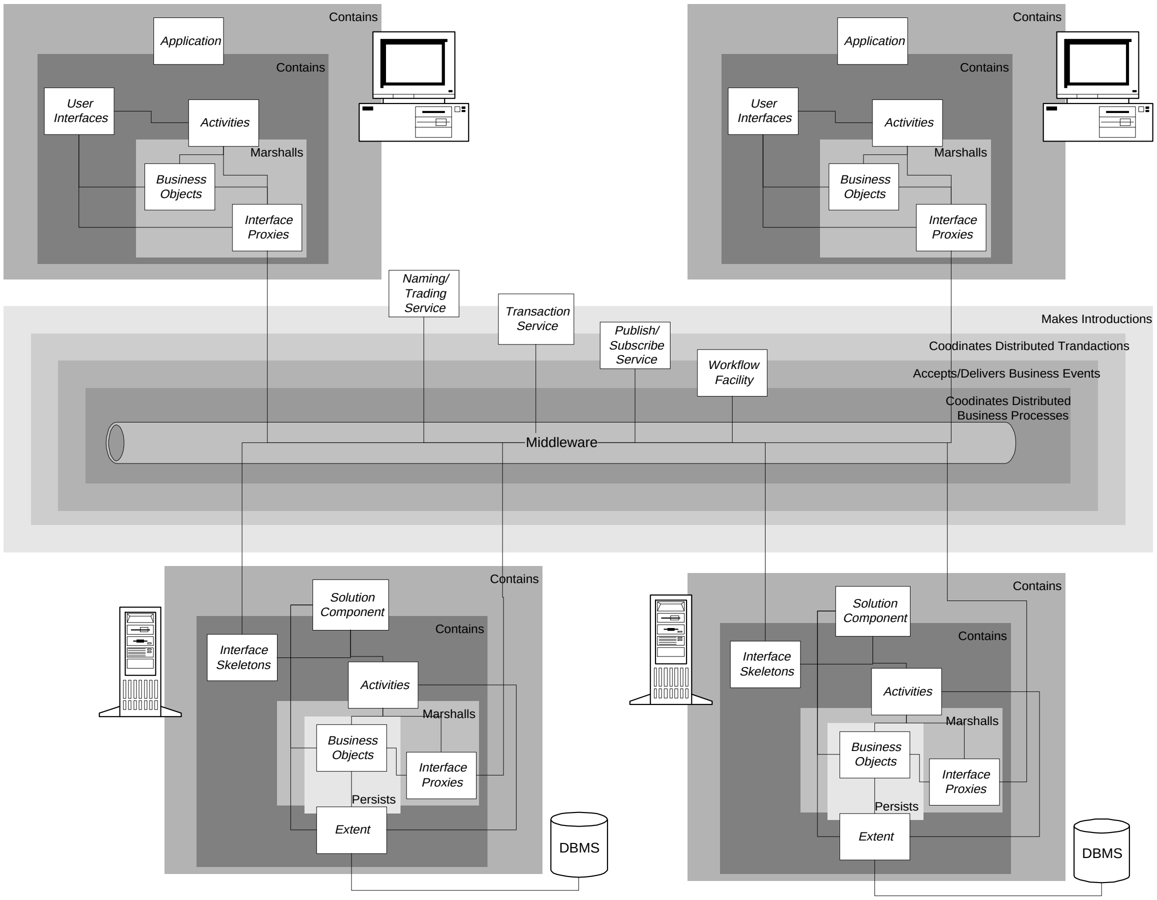
Frameworks for Integrating Components. This framework category helps the developer integrate components that potentially interact over a network, encourages partitioning, and helps deliver predictable performance for intercomponent messaging. It is made up of two subcategory frameworks:

- Workflow Facility
- Core Component Integration Services

Frameworks for Building Components. This framework category helps the developer build and introduce partitioning within a component. It is made up of several subcategories:

- User Interface Control
- Activity Control
- Business Class Modeling
- Business Class Persistence
- Component Interface
- Security
- IPS

EAS Metamodel



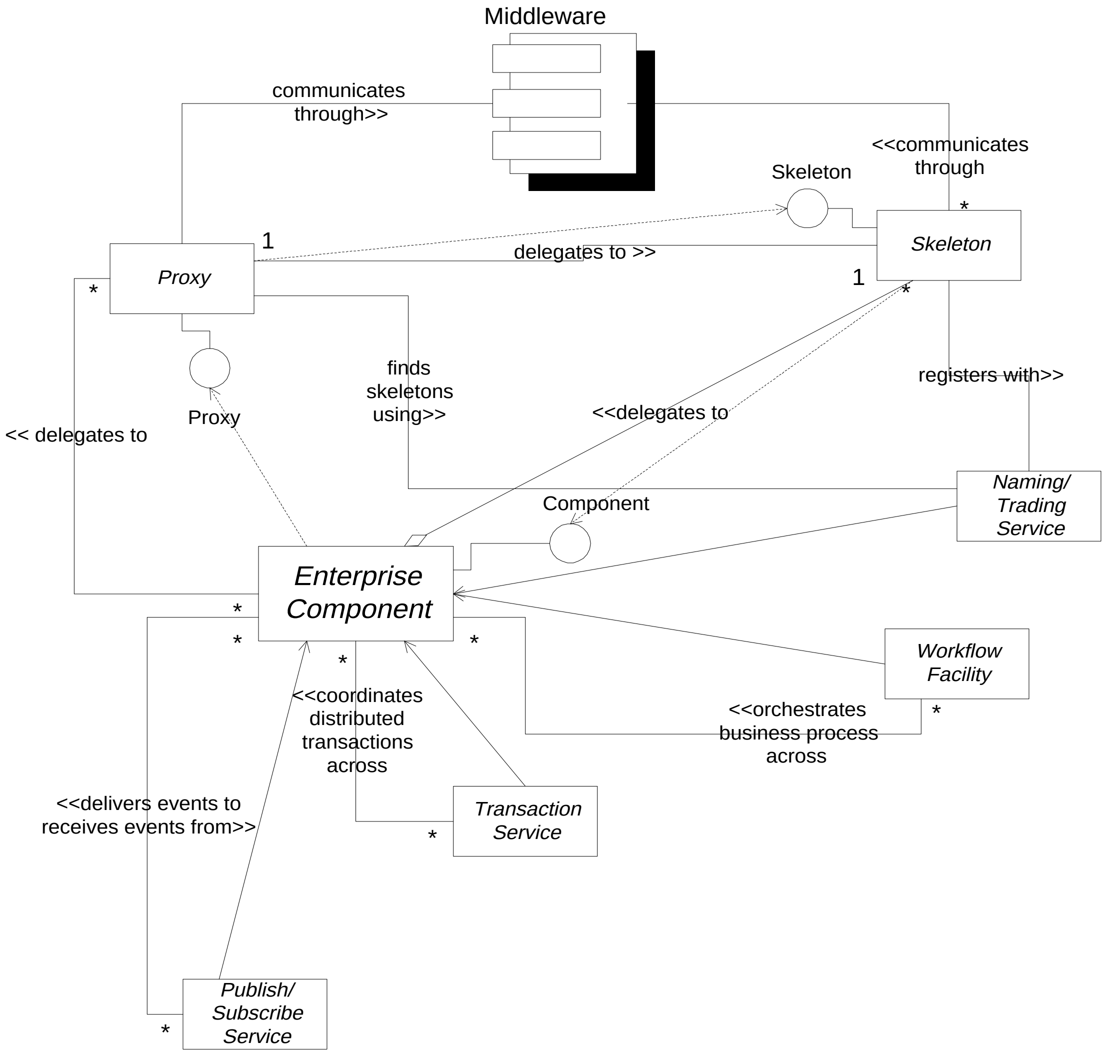
Frameworks for Integrating Components

Introduction

This framework category helps the developer integrate components that potentially interact over a network, encourages partitioning, and helps deliver predictable performance for intercomponent messaging. It is made up of two subcategory frameworks:

- Workflow Management
- Core Component Integration Services

Metamodel


Middleware
communicates through>>
<<communicates through
Skeleton
Skeleton
Proxy
1
*
1
*
*
delegates to >>
finds skeletons using>>
registers with>>
Proxy
<< delegates to
<<delegates to
Component
Naming/ Trading Service
Enterprise Component
*
*
*
*
Workflow Facility
*
<<orchestrates business process across
<<coordinates distributed transactions across
Transaction Service
*
<<delivers events to receives events from>>
Publish/ Subscribe Service
*

Workflow Management Framework Overview

Purpose

The Workflow Facility Framework externalizes significant state changes within a component solution and the policies for dealing with them. These state changes can span components and, thus, represent a set of integration points between Components. By externalizing these state changes and their policies, the Workflow Facility Framework centralizes these policies and models them to support monitoring and tailoring while the solution is in production. In addition, externalizing these policies from the individual functional components allows the management and enforcement of these policies using a sharable service, which supports easier integration of the functional components built using different technologies and patterns.

Services

Workflow Manager

The Workflow Manager maintains both the service's installed workflow definitions and executes all currently active workflow instances. Workflow definitions aggregate Workflow Transitions, which describe the conditions under which a state transition can be made and defines the component, component interface, and operation assigned to effect the state transition for the solution.

Subscription Manager

The Subscription Manager is responsible for managing a collection of Listeners. The Subscription Manager can create / reuse a Listener for every event that a workflow's transitions wish to react to. These listeners are created when the workflow is instantiated at registration time and are removed when the transition is completed. The Subscription Manager subscribes on the ORB on behalf of all its listeners and informs their related transitions when the events they are waiting for are received.

Listener

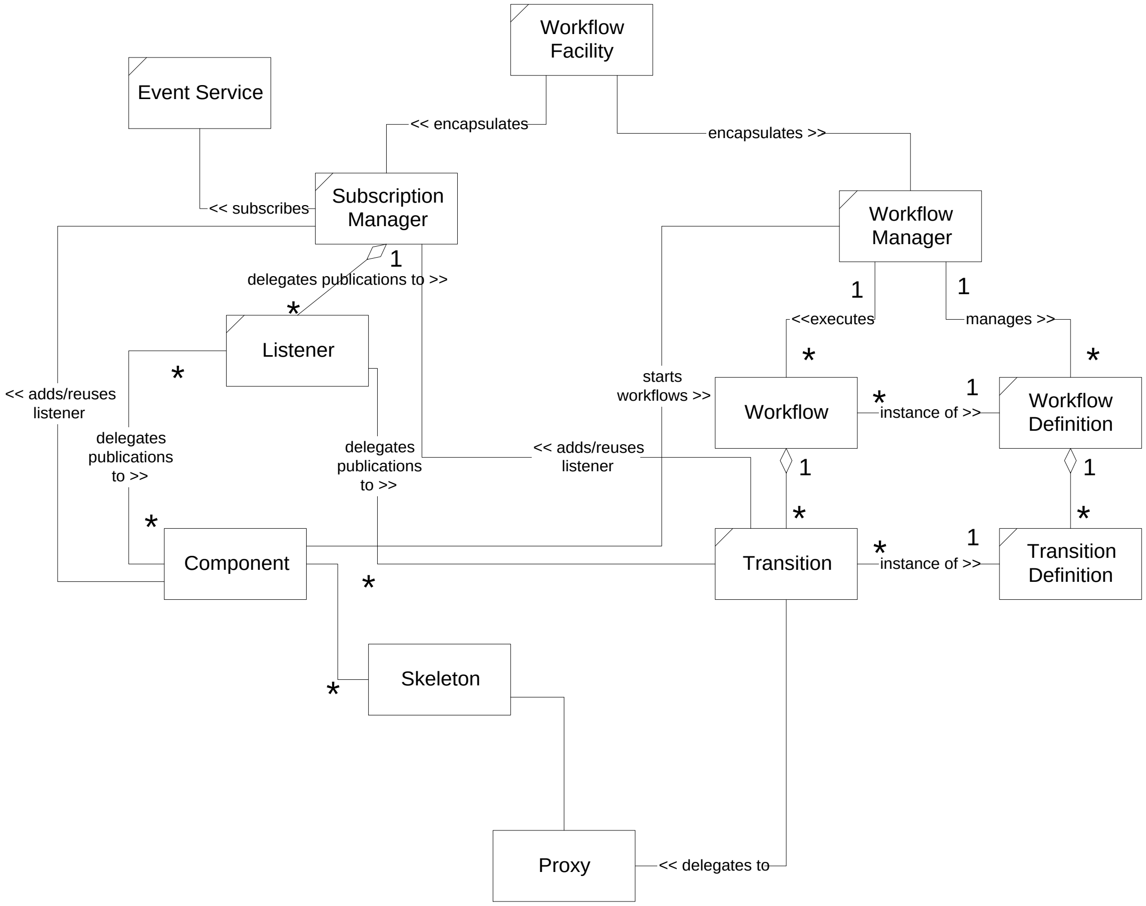
Listeners are created for every event that a component wishes to respond to within a particular context. When a Listener receives an event, the component is notified that an event was received for a particular context.

Workflow Facility

The Workflow Facility acts as the public interface into the entire Workflow framework. This service exposes the necessary interfaces which allow for the addition and removal of both Workflows and Workflow Definitions. This delegates to the Subscription Manager and the Workflow Manager.

Structure

Class Diagram



Participants

Workflow Facility

- Responsible for binding together the service's constituent manager classes
- Responsible for implementing public interfaces

Subscription Manager

- Manages collection of Listeners
- Creates Listener for each event to which a workflow's transition wishes to respond
- Creates Listener for each event to which a Component wishes to respond

- Subscribes to the ORB on behalf of all Listeners, informing related objects when events are received

Listener

- Created by the Subscription Manager, Listens multicasts publications received by the Subscription Manager to all related objects

Workflow Manager

- Maintains the Workflow Definitions
- Excecutes all currently active Workflow Instances

Workflow Definition

- Contains information required to specify the workflow
- Serve as templates for Workflow Instance creation by the Service

Workflow

- Defined by, and instantiated from, Workflow Definitions
- Guides the business object through a predefined Workflow
- Contains all necessary information to respond to events on the ORB
- May be dynamically altered while running

Workflow Transition Definition

- Contains information required to instantiate a Workflow Transition within a Workflow

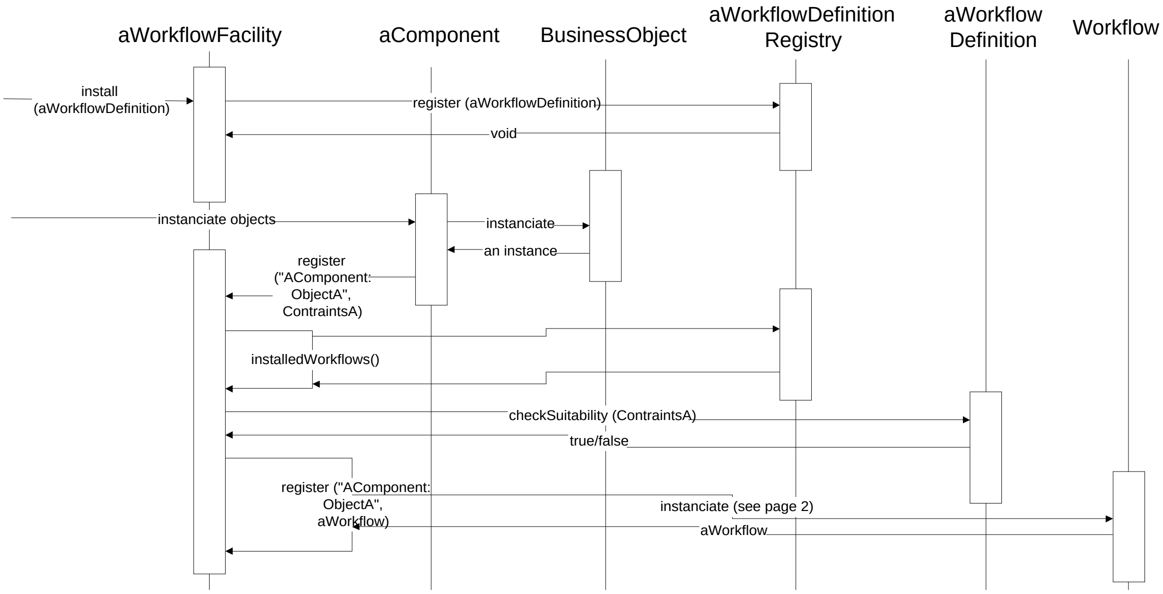
Workflow Transition

- Defined by, and instantiated from Workflow Transition Definitions
- Runtime objects owned by Workflows used to control some logical unit of work within a particular Life Cycle
- Invokes Component Interface operations to effect solution state transitions

Collaborations

Scenario:

The following diagram shows the flow of requests between the objects involved when workflows are installed with the workflow service manager.

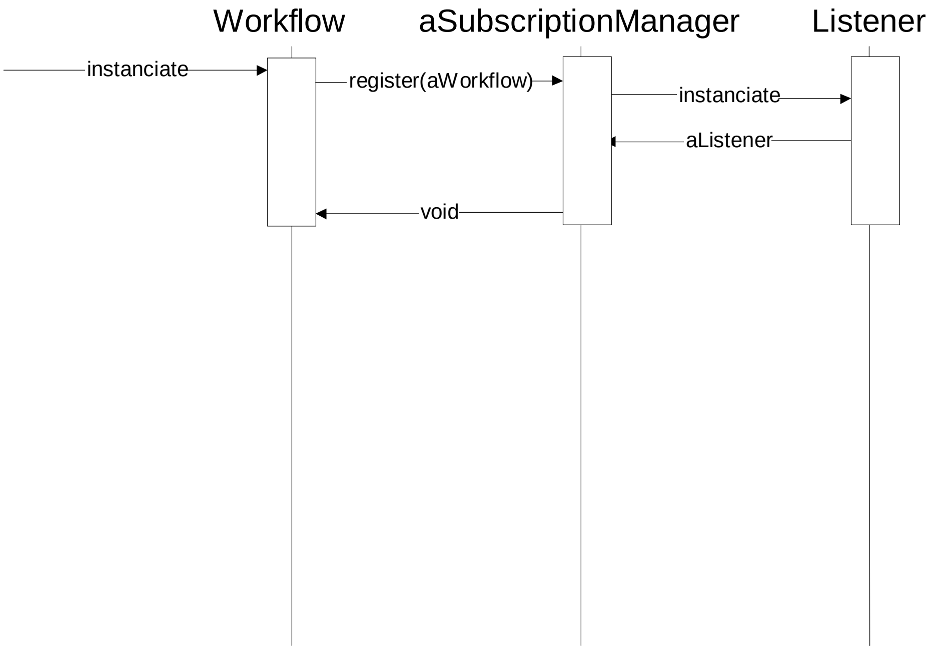

aWorkflowFacility
aComponent
BusinessObject
aWorkflowDefinition Registry
aWorkflow Definition
Workflow
install (aWorkflowDefinition)
register (aWorkflowDefinition)
void
instanciate objects
instanciate
an instance
register ("AComponent: ObjectA", ContraintsA)
installedWorkflows()
checkSuitability (ContraintsA)
true/false
register ("AComponent: ObjectA", aWorkflow)
instanciate (see page 2)
aWorkflow

Stimuli:

1. aWorkflowDefinition is installed into the Workflow Facility.
2. The definition is stored in the workflow registry.
3. The Component instanciates an instance of the Business Object.
4. The Component registers itself with the Workflow Facility, and requests that it find an appropriate workflow for the component based upon a set of constraints.
5. The Workflow Definition Registry is examined to find a suitable existing Workflow Definition that matches the set of provided constraints.
6. After finding a match, a new instance of a workflow is created and stored in the registry of active workflows.

Scenario:

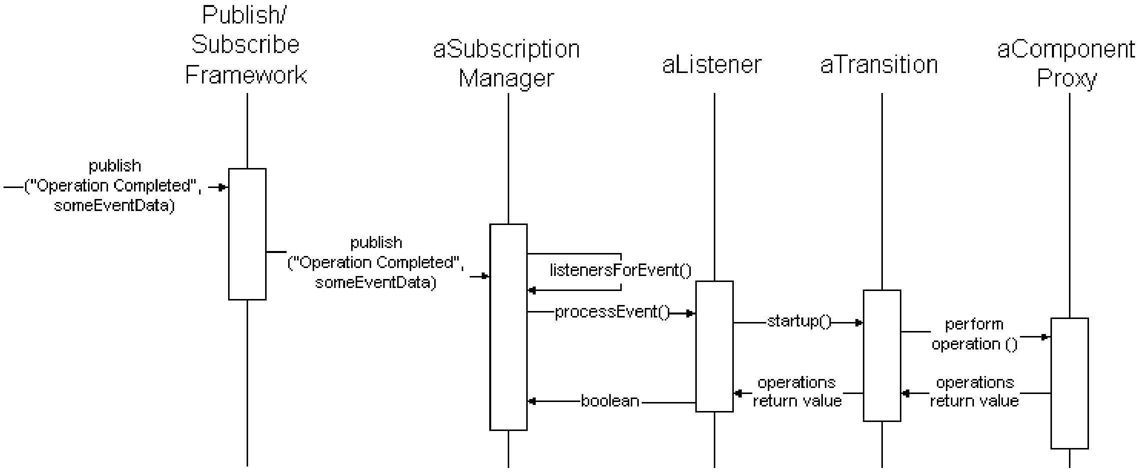
The following diagram shows the flow of requests between the objects involved when a workflow is instanciated and registered with the SubscriptionManager.



Stimuli:

1. A workflow is requested to be instantiated.
2. That new workflow registers itself with the Subscription Manager.
3. The Subscription Manager sets up a list of Listeners for all the event publications to which the newly instantiated workflow needs to subscribe.

Scenario:



Stimuli:

1. The "Operation Completed" event is posted to the Publish / Subscribe Framework for general consumption.
2. The Subscription Manager receives that event and asks its Listeners if they are interested in that event.
3. Once a Listener is found that is interested in that event, that Listener then tells its Transition to "startup."
4. The Transition tells its subject object, which is a component proxy, to execute that Transitions action.

Specification

```
//	This module specifies in OMG IDL the services of
//	the Workflow Facility.
//		Version 1.0
//		Revision History
//		Created 7/96
//		Modified 12/16/96
#ifndef __WORKFLOW_IDL
#define __WORKFLOW_IDL

// make async version for publish


module WorkflowFacility
{
	typedef sequence<string>			WorkflowConstraintList;
	typedef sequence<string>			WorkflowProperties;
	typedef sequence<string>
	WorkflowTransitionDefinitionEvents;

	enum WorkflowTransitionState {pending, active, complete};

	struct WorkflowTransitionDefinitionStructure
	{
		string					name;
		string					precondition;
		WorkflowTransitionDefinitionEvents		events;
		string					action;
		boolean						isRepeatable;
	};

	typedef sequence<WorkflowTransitionDefinitionStructure>
	WorkflowTransitionDefinitions;

	struct WorkflowDefinitionStructure
	{
		string					name;
		WorkflowProperties				properties;
		WorkflowTransitionDefinitions				transitions;
	};

	struct WorkflowTransitionInvocationStructure
	{
		long					startTime;
		long					endTime;
```

```
        long                                        transitionId;
        long                                        workflowId;
};

typedef sequence<WorkflowTransitionInvocationStructure>
WorkflowTransitionInvokations;
typedef sequence<any>
WorkflowTransitionParameters;

struct WorkflowTransitionStructure
{
        string                                      id;
        string                                      definitionName;
        string                                      workflow;
        string                                      subject;
        string                                      action;
        WorkflowTransitionParameters                        parameters;
        WorkflowTransitionState                             state;
        WorkflowTransitionInvokations                               invocations;
};

typedef sequence<WorkflowTransitionStructure>
WorkflowTransitions;


struct WorkflowStructure
{
        string                                      id;
        string                                      definitionName;
        WorkflowTransitions                         transitions;
        string                                      subject;
};

struct WorkflowListenerStructure
{
        string                                      workflowId;
        string                                      transitionId;
        string                                      event;
};

typedef sequence<WorkflowListenerStructure>                 ListenerList;
typedef sequence<WorkflowStructure>                         WorkflowList;
typedef sequence<WorkflowDefinitionStructure>
WorkflowDefinitionList;

exception
InvalidWorkflowDefinition{};
```

```
    exception
    WorkflowDefinitionAlreadyInstalled{};
    exception
    WorkflowDefinitionNotInstalled{};
    exception
    ComponentNotFound{};
    exception
    ObjectInterfaceNotFound{};
    exception
    WorkflowNotFound{};
    exception                                   NoActiveListeners{};
    exception
    NoActiveWorkflows{};
    exception
    NoInstalledWorkflows{};
    exception
    WorkflowNotActive{};
    exception                                   InvalidTransition{};
    exception
    WorkflowTransitionNotFound{};
    exception
    WorkflowTransitionActive{};

    // The service's core interface for installing
    // workflows and associating them with business
    // objects.
    interface WorkflowFacility
    {
        void install (in WorkflowDefinitionStructure defn)
            raises(InvalidWorkflowDefinition,
WorkflowDefinitionAlreadyInstalled);
        void remove (in string definitionName)
            raises(WorkflowDefinitionNotInstalled);
        string register(in string objectIdentifier, in WorkflowConstraintList
                componentConstraints)
            raises(ComponentNotFound, ObjectInterfaceNotFound,
WorkflowNotFound);
        void release(in string workflowId)
            raises(WorkflowNotFound);
        void stopWorkflow (in string workflowId)
            raises(WorkflowNotFound);
        WorkflowList activeWorkflows();
        WorkflowDefinitionList installedWorkflows();
    };

    // An interface providing access to the service's
    // currently active event listeners.
//  interface WorkflowSubscriptionManager
//      {
//              void activeListeners(in string workflowEvent, out ListenerList list)
//                      raises(NoActiveListeners);
//      };

        // An Interface providing facilities for monitoring
        // and managing installed and registered workflows
//      interface WorkflowManager
//      {
//              void activeWorkflows(out WorkflowList list)
//                      raises(NoActiveWorkflows);
//              void installedWorkflows(out WorkflowDefinitionList list)
//                      raises(NoInstalledWorkflows);
//              string currentTransition(in string workflowId)
//                      raises(WorkflowNotFound, WorkflowNotActive);
//              void addTransition(in long workflowId, in WorkflowTransition
transition)
//                      raises(WorkflowNotFound, InvalidTransition);
//              void removeTransition(in long workflowId, in long
workflowTransitionId)
//                      raises(WorkflowNotFound, WorkflowTransitionNotFound,
                                WorkflowTransitionActive);
//      };

};
#endif
```

Related Frameworks

Framework: Core Component Integration Services
Framework: Component Interface

Core Component Integration Services

These frameworks represent the basic frameworks and services necessary to build a solution from collaborating components. It is made up of several subcategory frameworks and services:

- *Component Specification/Construction Framework*. Component Specification Language - Extensions to IDL syntax to support definitions of components and their aggregated Interfaces.

- *Trader and Naming Services*. These services provide brokering between Components. The Trader Service adds Functional brokering constraints and load balancing / failover.

- *Event and Publish/Subscribe Service*. These services support the loose coupling of components. The Publish / Subscribe Service allows Components to subscribe to predefined events (or publications) with a filter defined in terms of publication attribute values. When another component publishes a publication whose attribute values match a subscriber's filter, the publication is delivered to the subscriber.

- *Transaction Service*. The Transaction Service provides transaction synchronization across the elements of a distributed component solution.

While these frameworks and services are expressed in terms of CORBA concepts and terms, these concepts can be applied in other non-CORBA environments (e.g., DCOM, DCE, RPC, etc.).

Component Specification - CSL Compiler

Component Specification Language (CSL) is a superset of the CORBA Interface Definition Language (IDL). Like IDL, CSL is a declarative language that allows you to specify interfaces. Unlike IDL, it also allows you to define components and specify which interfaces each component is going to provide and require. When a component provides an interface, it means that it implements the operation defined on that interface. When a component requires an interface, it means that it might need to invoke operations defined on that interface to carry out its business functions. A component can provide zero, one, or many interface(s) and can require zero, one, or many interfaces. Different components may provide or require the same interface.

Based on a CSL source file, a compiler would generate the IDL and implementation language source code that implements the specified interfaces and components. It is up to the developer to use the IDL, implementation language compilers/environments to compile and use the generated source code.

The CSL Syntax

Since CSL is a superset of IDL, you can refer to the CORBA specifications to obtain a complete description of the IDL syntax. Currently, CSL adds only three more keywords to the IDL grammar: *component*, *provides*, and *requires*. The component keyword is used to declare a component. The provides and requires keywords must be inside a component declaration and are used to specify which interfaces are required and provided by the component. Here is the general format:

```
component ComponentName
{
requires InterfaceName anyLabel;
provides InterfaceName anyLabel;
};
```

Naming Service Overview

Service Description

The Naming Service is the principle mechanism for locating components (and other exporters of interfaces) within a distributed environment.

The Naming Service enables the association of meaningful "names" to object references. It supports multiple, nested, and federated "names." Once a "name" is mapped to an object reference, other objects can use the "name" to obtain a reference and access the distributed object.

The Naming Service is a location-independent mechanism for locating objects. An object that binds a "name" to its object reference in the Naming Service is now accessible through that "name". The requesting (or client) object need not know or care where the "server" object is located on the network. It can use the "name" to obtain an object reference from the Naming Service regardless of the "server" object's location.

Once an object reference is obtained, the client object can talk directly to the "named" object.

Services

Meaningful Names

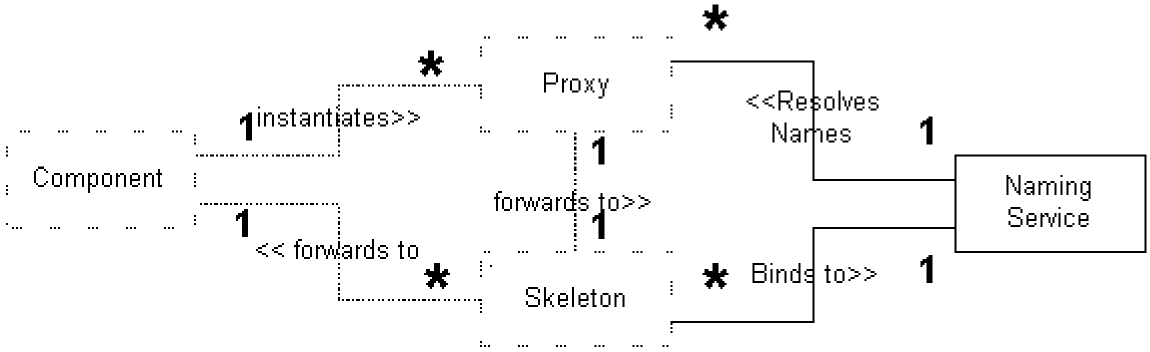
The Naming Service allows the association of a meaningful name with an object reference. A meaningful name is much easier for a human software developer to use than an object reference. Object references are a lot more difficult for programmers to reference.

Location Transparency

The Naming Service is a location independent mechanism for locating objects. An object that binds a "name" to its object reference in the Naming Service, is now accessible through a "name". The requesting (or client) object need not know or care where the "server" object is located on the network. It can use the "name" to obtain an object reference from the Naming Service regardless of the "server" object's location.

Structure

Class Diagram



Participants

Component

- Encapsulates business logic
- Holds skeletons for all services
- Holds Proxies to other components skeletons

Proxy

- Transparently connects to interfaces on other components
- Models an Importer’s view of an "interface"

Skeleton

- Receives requests from proxies
- Models an Exporter’s view of an "interface"

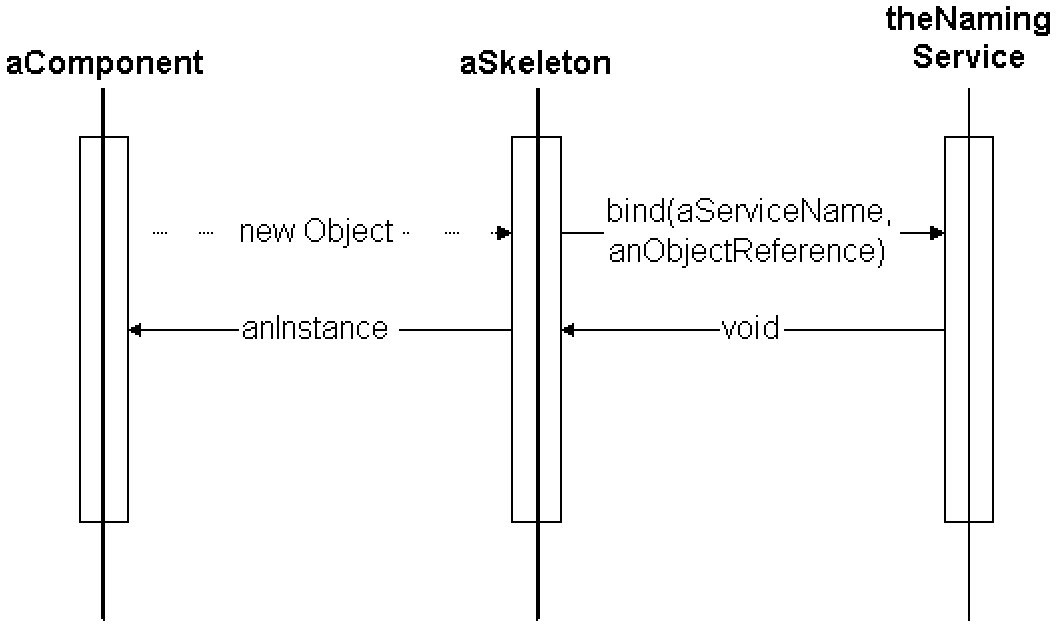
Naming Service

- Maintains a list of "names" and associated object references
- Returns an object reference for a "name"

Collaborations

Scenario:

The following diagram shows the registering of an object with the Naming Service.

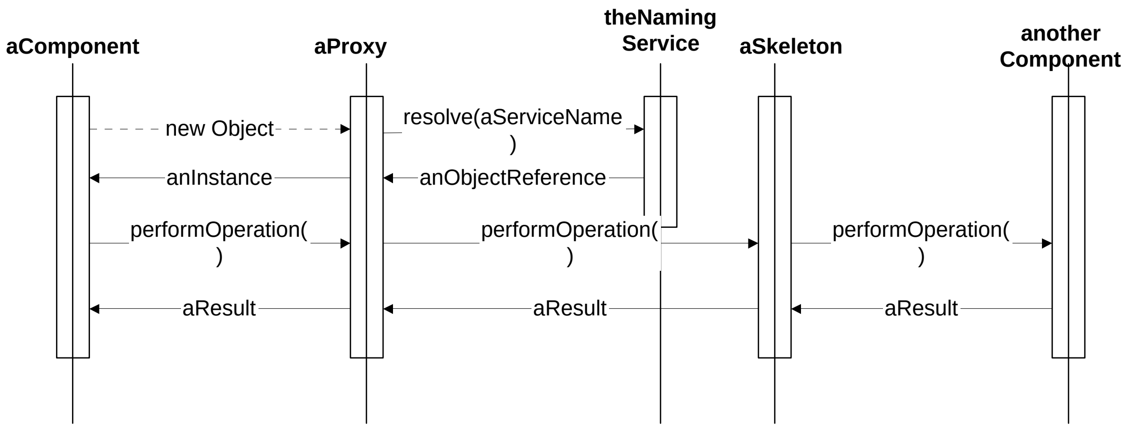


Stimuli:

1. aComponent is instantiated. AComponent instantiates aSkeleton.
2. aSkeleton registers (binds) with the Naming Service.
3. The Naming Service saves the serviceName and anObjectReference to the Skeleton.

Scenario:

The following diagram depicts a client requesting a service from the Naming Service followed by a request to the service.

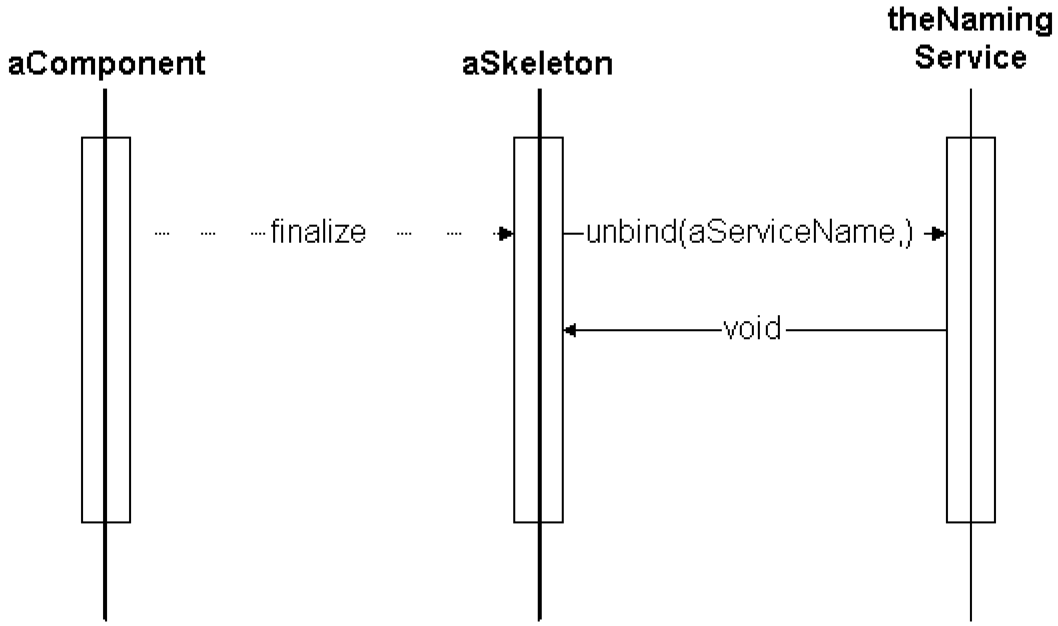


Stimuli:

1. aComponent is instantiated. AComponent instantiates aProxy.
2. aProxy asks (resolves) the Naming Service for a reference to a service (represented by aServiceName).
3. The Naming Service returns anObjectReference (to aSkeleton) to aProxy.
4. aProxy saves anObjectReference for future use.
5. aComponent makes a performOperation() request of aProxy.
6. aProxy uses anObjectReference to make a performOperation() request of aSkeleton.
7. aSkeleton passes the performOperation() request to anotherComponent.
8. anotherComponent executes performOperation() and returns aResult through aSkeleton and aProxy to aComponent.

Scenario:

The following diagram depicts a Component unregistering a service.



Stimuli:

1. aComponent no longer wishes to provide a service. It removes all references to aSkeleton. The skeleton is eventually garbage collected.
2. aSkeleton withdraws (unbinds) the service it had been providing.
3. The Naming Service removes aServiceName from its list of services.

Specification

Package: CORBA Service Specifications

Related Frameworks

Framework: Component Interface Framework Overview

Framework: Trader Service Overview

Trader Service Overview

Service Description

The Trader Service is another mechanism (along with the Naming Service) for locating components within a distributed environment. It is a service through which components can advertise their capabilities and other components can match their needs against these advertised capabilities.

The Trader Service facilitates dynamic discovery of, and late binding to objects. Service providers give the Trader Service a description of their service together with an object reference to an interface (implemented as a skeleton) to the described service. Clients (anything requesting a service) ask the Trader for a service having certain characteristics. The Trader matches the request with the list of currently registered object references and returns a list of matches. The requester can then directly contact the service provider.

Services

Location Transparency

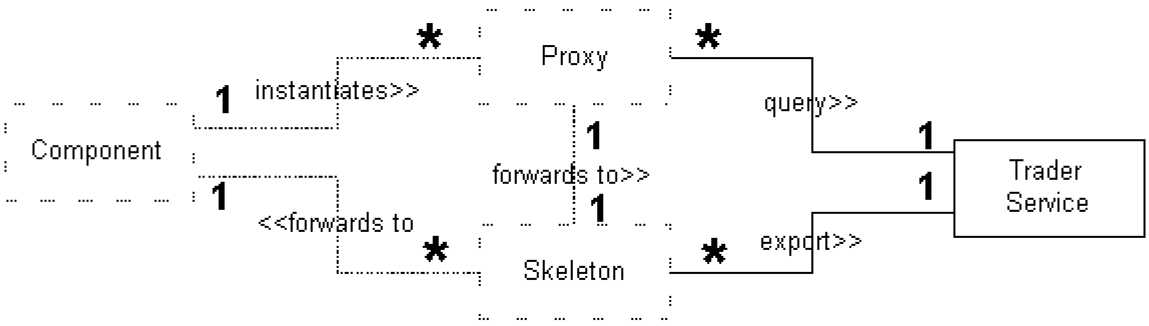
The Trader Service is a location independent mechanism for locating objects. Once an object has registered (exported) itself with the Trader Service, it is now accessible through its service type and properties. As with a Naming Service, the requesting (or client) object need not know or care where the "server" object is located on the network.

Dynamic Discovery of Service Providers

The Trader Service facilitates dynamic discovery of, and late binding to objects. A client object requests a particular service from the Trader Service. The Trader Service will locate a suitable service and return an object reference. The client object can now bind and communicate with the service.

Structure

Class Diagram



Participants

Component

- Encapsulates business logic
- Holds skeletons for all services
- Holds Proxies to other components interfaces

Proxy

- Transparently connects to interfaces on other components
- Models an importer’s view of an "interface"

Skeleton

- Receives requests from proxies
- Models an exporter’s view of an "interface"

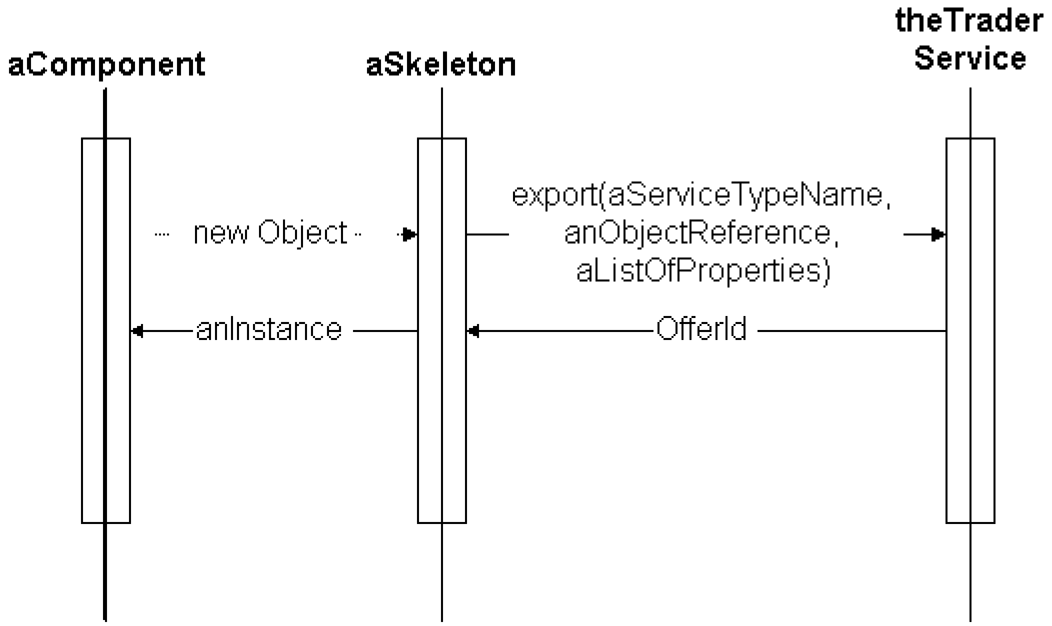
Trader Service

- Maintains a list of service types, their properties, and associated object references
- Matches requests and associated constraints with services and their associated properties
- Returns object references to skeletons providing a service

Collaborations

Scenario:

The following diagram shows the registering of an object with the Trader Service.

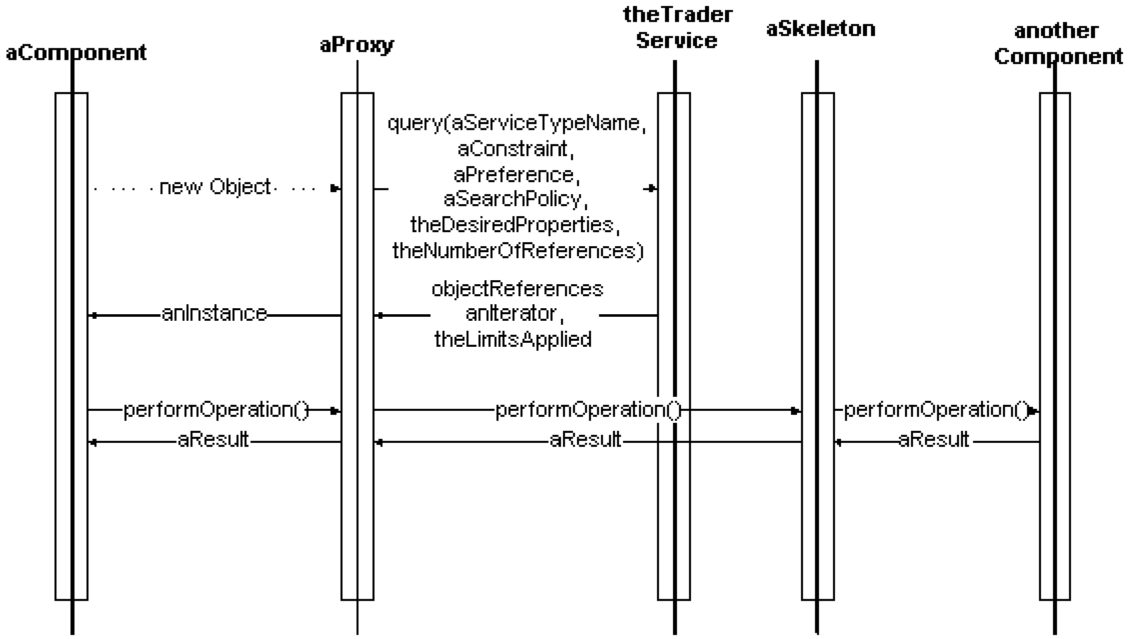


Stimuli:

1. aComponent is instantiated.
2. aSkeleton is instantiated.
3. aSkeleton registers (exports) with the Trader Service.
4. The Trader Service saves the serviceTypeName, anObjectReference to the interface and a list of associated properties.

Scenario:

The following diagram depicts a client requesting a service from the Trader Service followed by a request of that service.

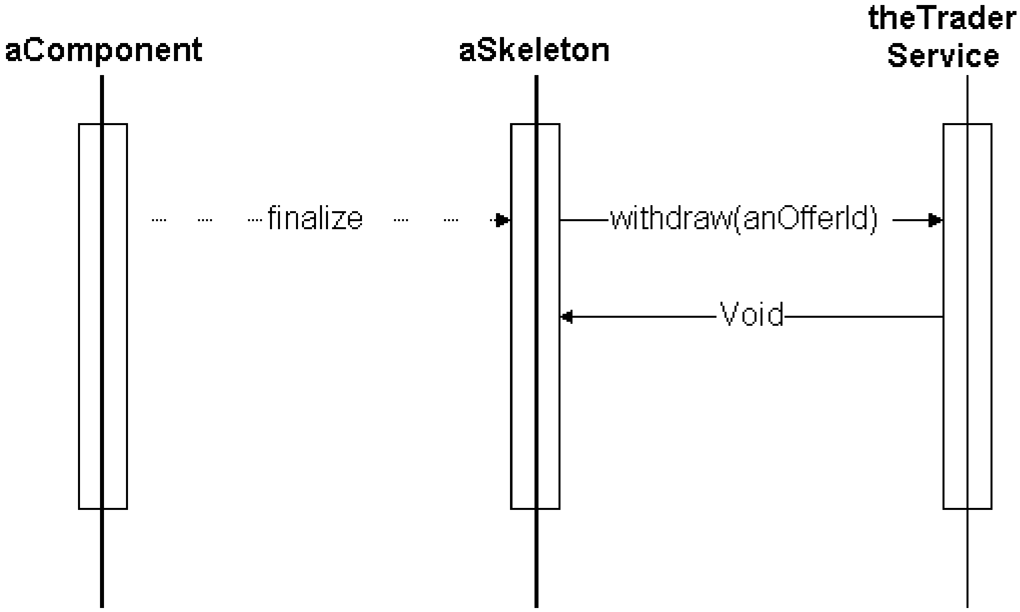

aComponent
aProxy
theTrader Service
aSkeleton
another Component
new Object
query(aServiceTypeName, aConstraint, aPreference, aSearchPolicy, theDesiredProperties, theNumberOfReferences)
anInstance
objectReferences anIterator, theLimitsApplied
performOperation()
performOperation()
performOperation()
aResult
aResult
aResult

Stimuli:

1. aComponent instantiates aProxy.
2. aProxy queries the Trader Service for a service that meets its constraints.
3. The Trader returns a list of services (objectReferences) that meet the constraints.
4. aProxy saves the first objectReference (the objectReferences should be ordered according to my specified criteria) for future use.
5. aComponent makes a performOperation() request of aProxy.
6. aProxy uses the objectReference to make a performOperation() request of aSkeleton.
7. aSkeleton passes the peformOperation() request to anotherComponent.
8. anotherComponent executes performOperation() and returns aResult through aSkeleton and aProxy to aComponent.

Scenario:

The following diagram depicts a Component unregistering a service.



Stimuli:

1. aComponent no longer wishes to provide a service. It removes all references to aSkeleton. The interface is eventually garbage collected.
2. aSkeleton withdraws the service it had been providing.

Specification

Package: CORBA Service Specifications

Related Frameworks

Framework: Component Interface Framework Overview

Framework: Naming Service Overview

Event Service Overview

Service Description

An Event Service allows objects (usually component objects) to dynamically register or unregister an interest in a specific event. An event is an "occurrence" within a component (or object) specified to be of interest to one or more objects. A "notification" is a message an object sends to interested parties informing them that a specific event occurred. The Event Service decouples communication between the components (or objects) such that the event supplier (object sending an event) need not know the event consumer (object or objects interested in the event).

The Push model

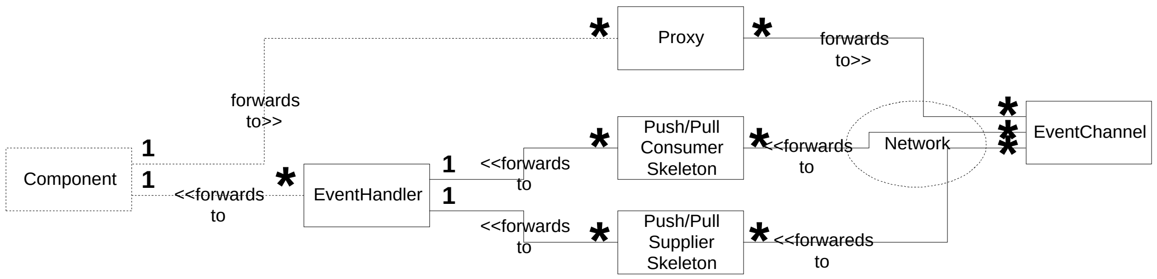
The push model relies upon suppliers "pushing" events to consumers. Thus, an event supplier "pushes" an event onto an event channel and the event channel "pushes" the event to the consumer object.

The Pull model

The pull model relies upon consumer polling or "pulling" events from a supplier. Thus, the consumer "pulls" an event from an event channel and the event channel "pulls" the event from the supplier.

Structure

Class Diagram



Participants

Component

- Encapsulates business objects interested in publishing and subscribing

EventHandler

- Implements one or more of the Push / Pull Supplier / Consumer interfaces
- Ties the interface to the component

Proxy

- Used to communicate with the Event Channel
- Used to communicate with all supplierProxy and consumerProxy objects encapsulated in the Event Channel

Push/Pull Consumer Skeleton

- Receives events for Consumers using the PUSH model
- Receives Event Channel disconnect notice for Consumers using the PULL Model

Push/Pull Supplier Skeleton

- Receives Event Channel disconnect notice for Suppliers using the PUSH Model
- Receives requests for event for Suppliers using the PULL model

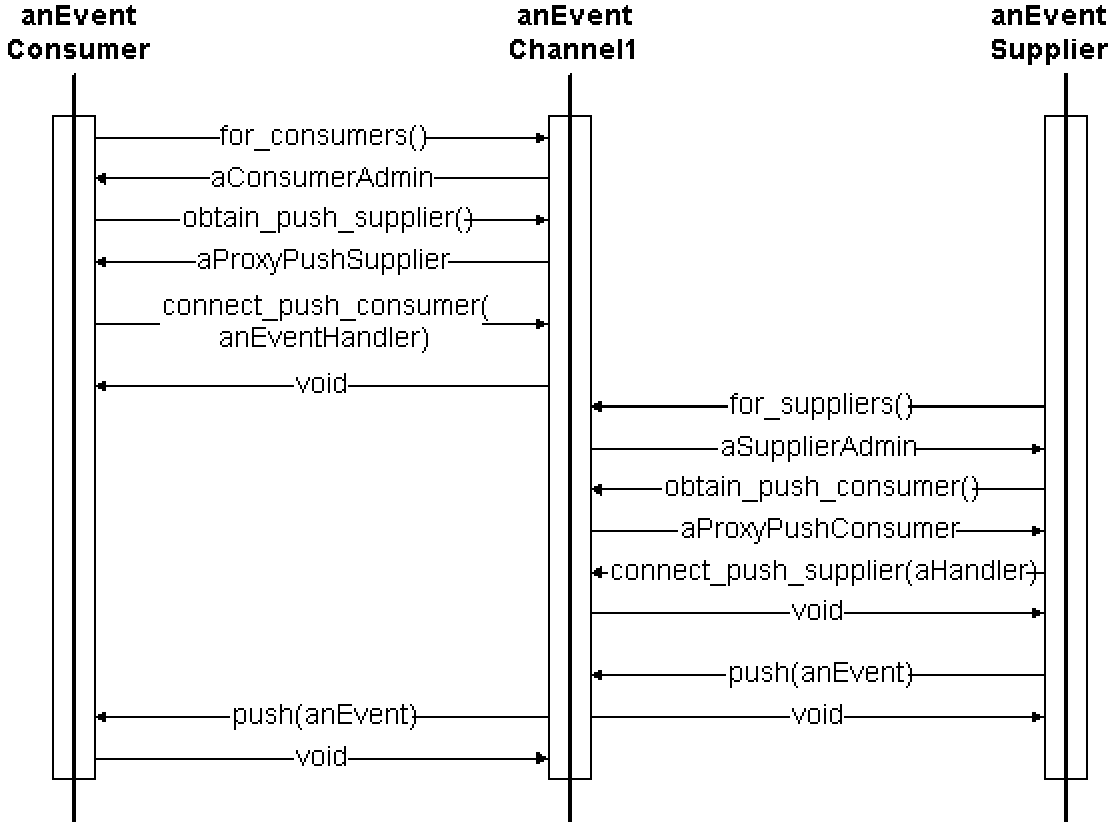
Event Channel

- Decouples the suppliers and consumers
- Manages the consumers of events
- Forwards

Collaborations

Scenario:

Registering a consumer and supplier to receive and supply anEvent using the PUSH method.



Stimuli:

Registering an event consumer

1. Send for_consumers() to anEventChannel1 and get aConsumerAdmin.
2. Send obtain_push_supplier to anEventChannel1 (using aConsumerAdmin interface) to create aProxyPushSupplier on anEventChannel1. aProxyPushSupplier will supply anEventConsumer with events.
3. Return an interface to the aProxyPushSupplier.
4. Send connect_push_consumer with anEventHandler to anEventChannel1. This registers anEventHandler as the object to receive incoming events.

Registering an event supplier

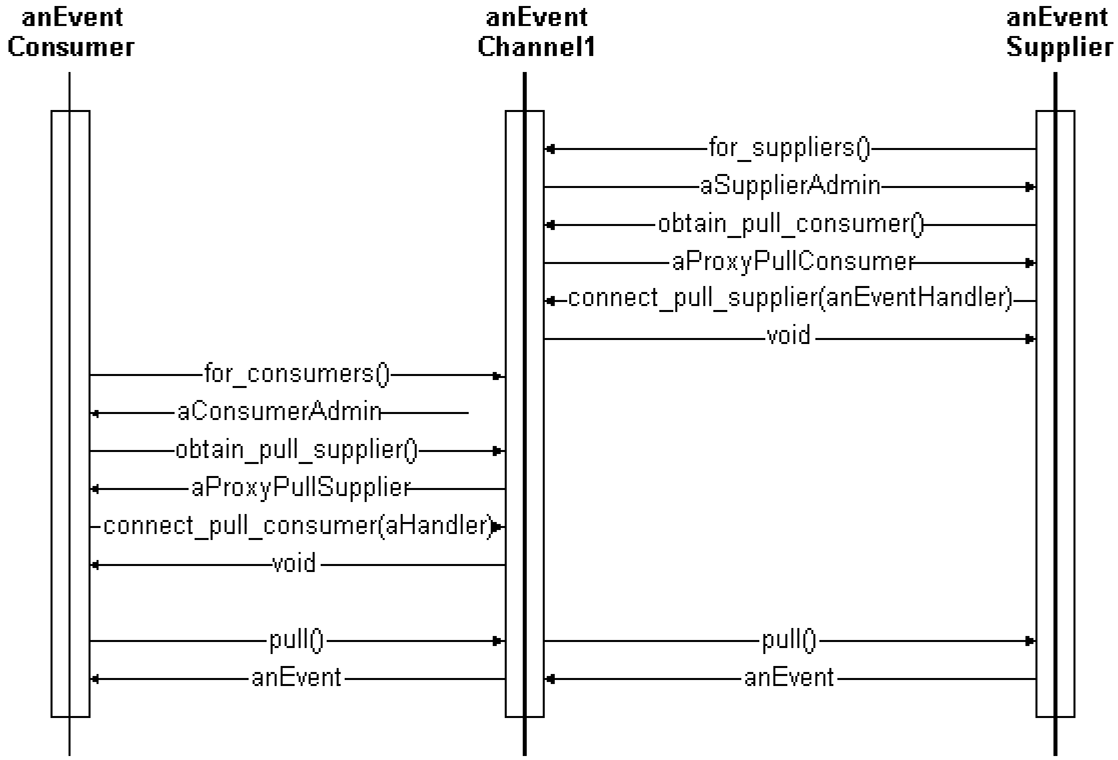
1. Send for_suppliers() to anEventChannel1 and get aSupplierAdmin.
2. Send obtain_push_consumer to anEventChannel1 (using aSupplierAdmin interface) to create aProxyPushConsumer on anEventChannel1. aProxyPushConsumer will consume events sent by anEventSupplier.
3. Return an interface to aProxyPushConsumer.
4. Send connect_push_supplier with aHandler to anEventChannel1. This registers aHandler to receive "disconnect" messages when anEventChannel1 is shutdown.

Pushing an event

1. anEvent occurs on anEventSupplier. anEventSupplier pushes anEvent to anEventChannel1 and is returned a void.
2. anEventChannel1 pushes anEvent to all registered consumers. In this case, anEventChannel1 pushes anEvent to anEventConsumer.

Scenario:

Registering a consumer and supplier to receive and supply anEvent using the PULL method.



Stimuli:

Registering an event supplier

1. Send for_suppliers() to anEventChannel1 and get aSupplierAdmin.
2. Send obtain_pull_consumer to anEventChannel1 (using aSupplierAdmin interface) to create aProxyPullConsumer on anEventChannel1. aProxyPullConsumer will consume events from anEventSupplier.
3. Return an interface to the aProxyPullConsumer.
4. Send connect_pull_supplier with anEventHandler to anEventChannel1. This registers anEventHandler to receive requests for events from anEventChannel1.

Registering an event consumer

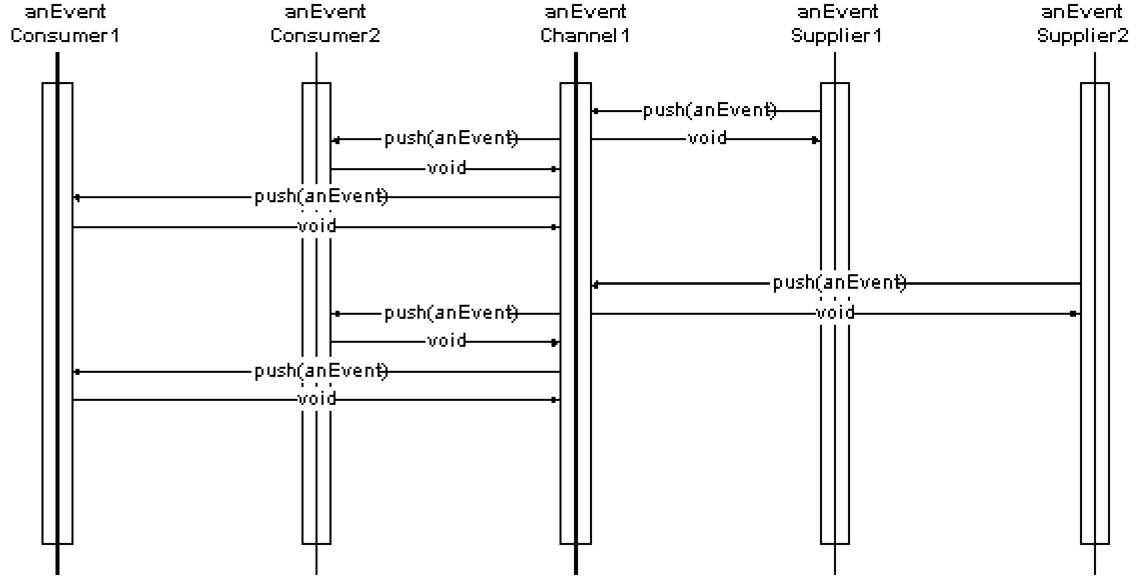
1. Send for_consumers() to anEventChannel1 and get aConsumerAdmin.
2. Send obtain_pull_supplier to anEventChannel1 (using aConsumerAdmin interface) to create aProxyPullSupplier on anEventChannel1. aProxyPullSupplier will supply anEventConsumer with events.
3. Return an interface to the aProxyPullSupplier.
4. Send connect_pull_consumer with anEventHandler to anEventChannel1. This registers aHandler to receive disconnect messages when anEventChannel1 shuts down.

Pulling an event

1. anEventConsumer would like to know when the next occurrence of anEvent happens.
2. anEventConsumer sends a pull() message to anEventChannel1 and blocks until anEvent occurs (try_pull() won't block).
3. anEventChannel1 sends a pull() to anEventSupplier. anEventSupplier returns anEvent.
4. anEventChannel1 returns anEvent to anEventConsumer.

Scenario:

Pushing anEvent from multiple suppliers to multiple consumers. This scenario assumes that all consumers and suppliers are already registered with anEventChannel1.



Stimuli:

1. anEvent occurs on anEventSupplier1. anEventSupplier pushes anEvent to anEventChannel1 and is returned a void.
2. anEventChannel1 pushes anEvent to anEventConsumer1.
3. anEventChannel1 pushes anEvent to anEventConsumer2.
4. anEvent occurs on anEventSupplier2. anEventSupplier pushes anEvent to anEventChannel1 and is returned a void.
5. anEventChannel1 pushes anEvent to anEventConsumer1.
6. anEventChannel1 pushes anEvent to anEventConsumer2.

Specification

Package: CORBA Service Specifications

Related Frameworks

Framework: Component Interface Framework Overview

Framework: Publish Subscribe Service Overview

Publish/Subscribe Service Overview

Service Description

The Publish Subscribe Service extends the functionality of a push style Event Service. It is built upon an Event Service and provides the same decoupling mechanism as an Event Service, yet it provides more.

The Publish Subscribe Service uses trader functionality (not the Trader Service) to provide filtering. Event consumers can register an interest in an event with a set of properties. Event suppliers can send events with a set of constraints. The Publish Subscribe Service will match the consumers (and their properties) with a supplier's event (and its constraints) to determine the consumers that should receive this event. It will then push the event to those consumers. This gives event suppliers more control over the consumers of their events and vice versa.

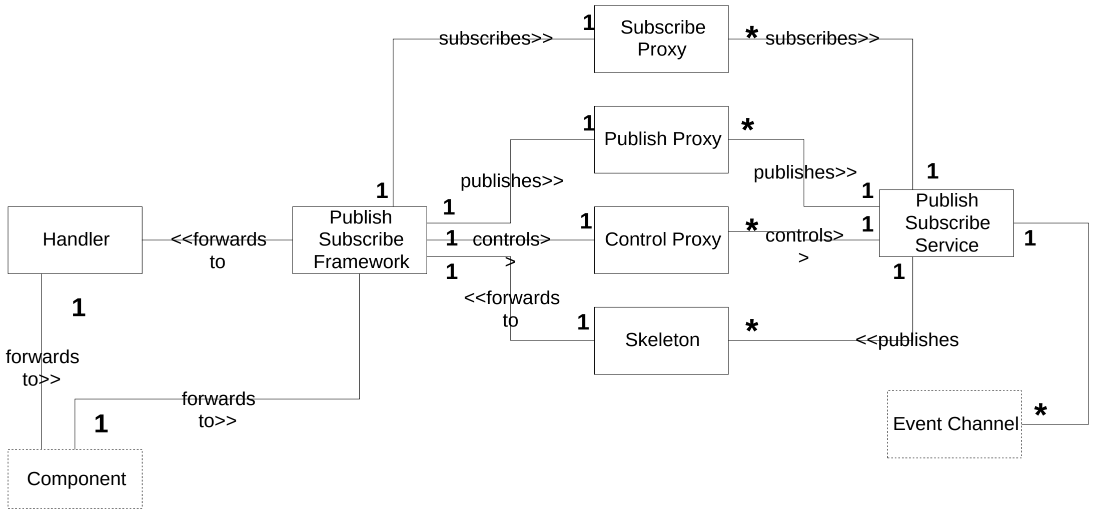
The Publish Subscribe Service adds support for a publication semantic. A Publication Semantics allow an event supplier to more specifically determine the consumers of an event. In some instances, it is desirable to send an event to one consumer as opposed to all consumers. A publication semantic of ONE or BEST provides this functionality. The publication semantics are as follows:

- ALL - Guaranteed delivery to all subscribers whether or not they are currently running
- ACTIVE - Publish to all subscribers currently "active" or available
- BEST - Publish to the one "best" subscriber
- ONE - Publish to only one subscriber

The Publish Subscribe Service provides guaranteed delivery of events. It utilizes a persistent store to maintain events and will deliver the event to a consumer at a future time.

Structure

Class Diagram



Participants

Component

- Encapsulates business objects interested in publishing and subscribing

Publish Subscribe Framework

- Facilitates publishing and subscribing within the executable
- Performs all the processing
- Publishes to the Publish Subscribe Service
- Subscribes with the Publish Subscribe Service on behalf of the component
- Delivers publications within the executable

Subscriber Proxy

- Used to subscribe with the Publish Subscribe Service

Publisher Proxy

- Used to publish with the Publish Subscribe Service

Control Proxy

- Used to notify the Publish Subscribe Service when the Framework is up or down

Skeleton

- Receives all publications from the Publish Subscribe Service

Handler

- Ties the Publish Subscribe Framework to the Component
- Delivers the publication to the appropriate component or object within a component

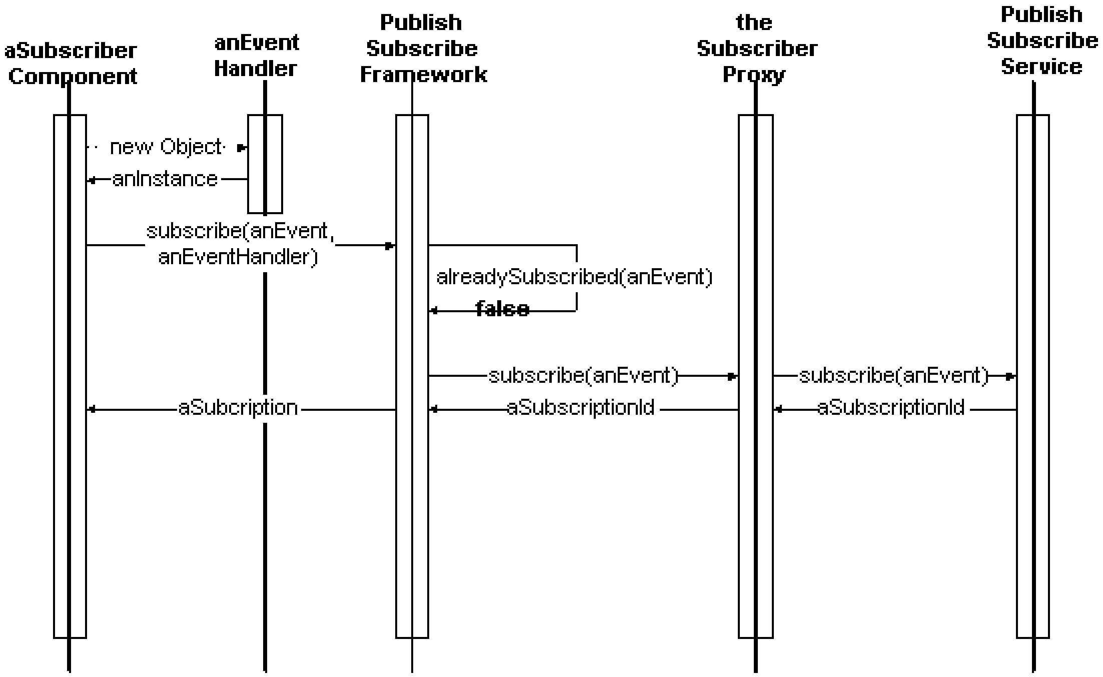
Publish Subscribe Service

- Manages subscribers to a particular event
- Receives publications to events
- Matches publications with subscribers and delivers events

Collaborations

Scenario:

Subscribing to anEvent for the first time.

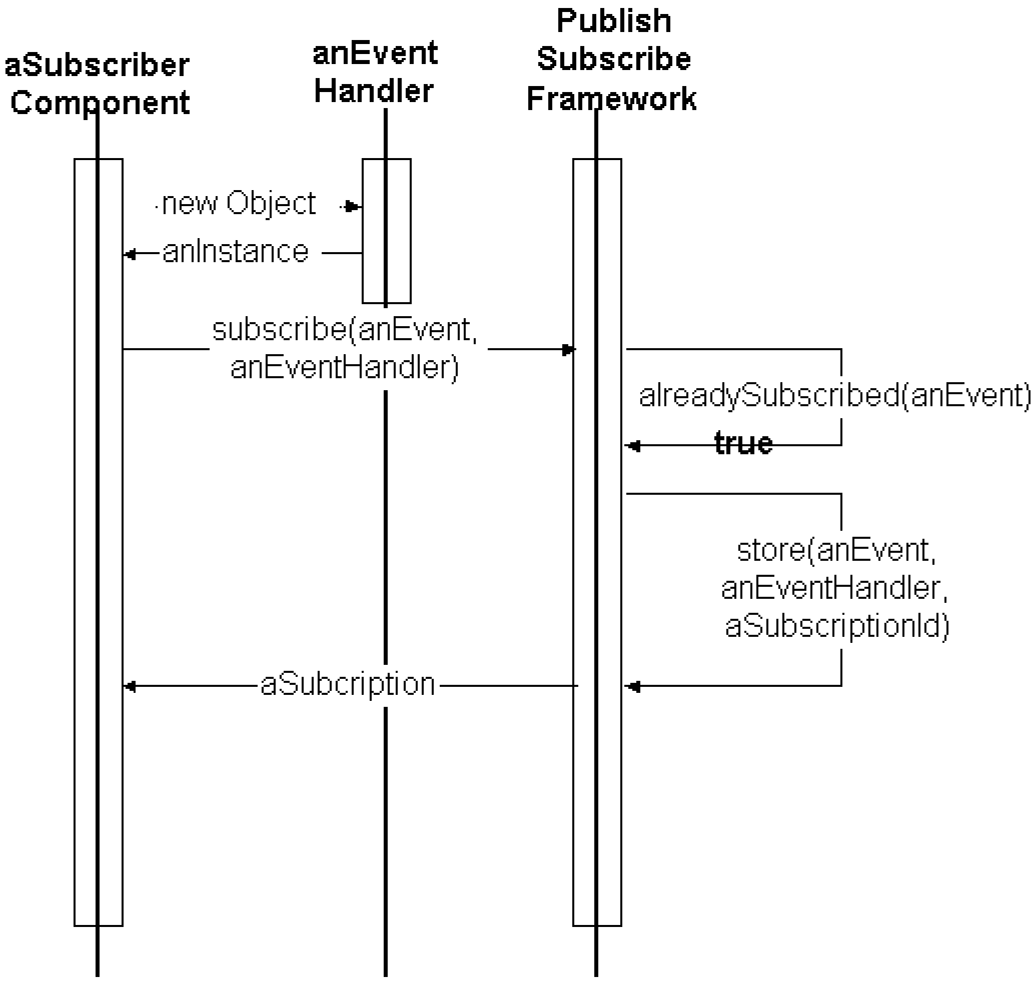


Stimuli:

1. aSubscriberComponent creates anEventHandler. anEventHandler is specific to this subscription and knows how to deliver to aSubscriberComponent.
2. aComponent subscribes to anEvent with anEventHandler. The message is sent to the PublishSubscribeFramework.
3. The PublishSubscribeFramework checks if we have alreadySubscribed() to this particular event.
4. Since we haven't alreadySubscribed, the PublishSubscribeFramework subscribes with the Publish Subscribe Service using theSubscriberProxy.
   (See next scenario for an example where we had already subscribed)
5. The Publish Subscribe Service returns aSubscriptionId. This id is used for future references to this particular subscription.
6. The PublishSubscribeFramework stores anEvent, anEventHandler and aSubsciptionId.
7. The PublishSubscribeFramework creates aSubscription (containing aSubscriptionId and anEvent) and returns it to aSubscriberComponent.

Scenario:

Subscribing to anEvent for the second time in the same executable.


aSubscriber Component
anEvent Handler
Publish Subscribe Framework
new Object
anInstance
subscribe(anEvent, anEventHandler)
alreadySubscribed(anEvent)
true
store(anEvent, anEventHandler, aSubscriptionId)
aSubcription

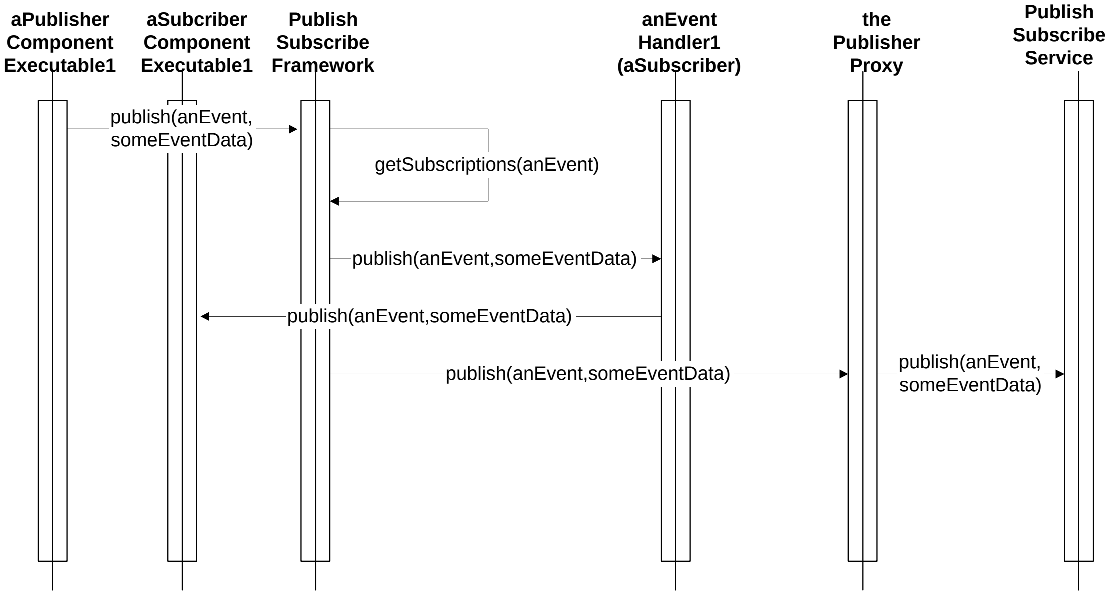
Stimuli:

1. aSubscriberComponent creates anEventHandler. anEventHandler is specific to this subscription and knows how to deliver to aSubscriberComponent.
2. aComponent subscribes to anEvent with anEventHandler. The message is sent to the PublishSubscribeFramework.
3. The PublishSubscribeFramework checks if it has alreadySubscribed() to this particular event.
4. Since we have alreadySubscribed, the PublishSubscribeFramework stores anEvent, anEventHandler and the returned aSubsciptionId.

5. The PublishSubscribeFramework creates aSubscription (containing aSubscriptionId and anEvent) and returns it to aSubscriberComponent.

Scenario:

Publishing anEvent and delivering anEvent to subscribers within the current executable.

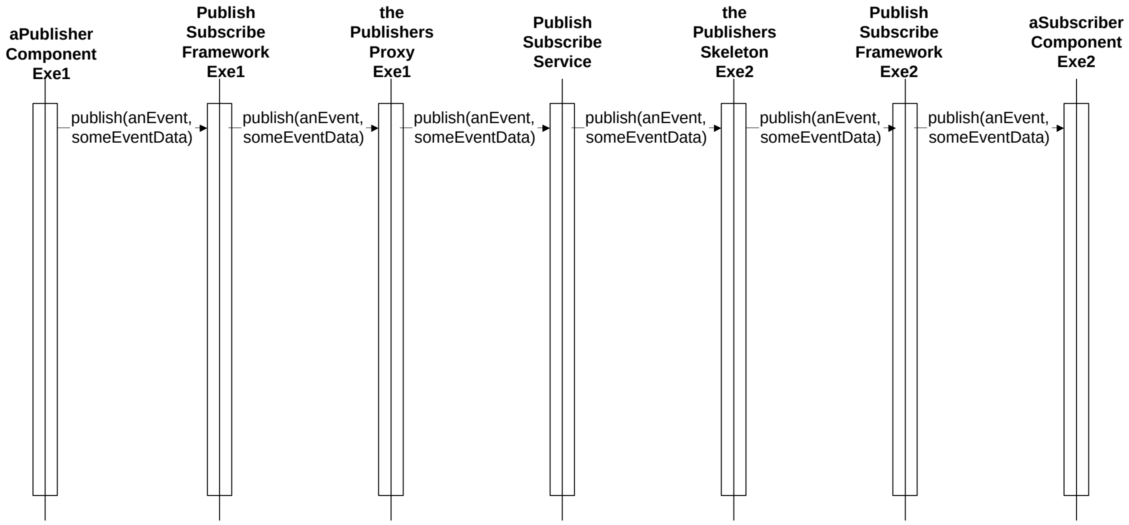

aPublisher Component Executable1
aSubcriber Component Executable1
Publish Subscribe Framework
anEvent Handler1 (aSubscriber)
the Publisher Proxy
Publish Subscribe Service
publish(anEvent, someEventData)
getSubscriptions(anEvent)
publish(anEvent,someEventData)
publish(anEvent,someEventData)
publish(anEvent,someEventData)
publish(anEvent, someEventData)

Stimuli:

1. aPublisherComponent publishes anEvent with someEventData. The message is sent to the EventServiceFramework.
2. The PublishSubscribeFramework performs an internal getSubscriptions with anEvent.
3. anEventHandler1 is returned.
4. The PublishSubscribeFramework publishes anEvent and someEventData to anEventHandler1.
5. anEventHandler1 publishes anEvent and someEventData to aSubscriberComponentExecutable1.
6. The PublishSubscribeFramework publishes anEvent and someEventData to the PublishSubscribeService using thePublisherProxy.
7. The PublishSubscribeFramework will now deliver the publication to subscribers in other executables (other than Executable1).

Scenario:

Publishing anEvent. Delivering anEvent to a subscriber in another executable.


aPublisher Component Exe1
Publish Subscribe Framework Exe1
the Publishers Proxy Exe1
Publish Subscribe Service
the Publishers Skeleton Exe2
Publish Subscribe Framework Exe2
aSubscriber Component Exe2
publish(anEvent, someEventData)
publish(anEvent, someEventData)
publish(anEvent, someEventData)
publish(anEvent, someEventData)
publish(anEvent, someEventData)
publish(anEvent, someEventData)

Stimuli:

Note: The details associated with delivering events within the PublishSubscribeFramework are shown in the previous scenario, and not in this scenario.

1. aPublisherComponentExe1 publishes anEvent with someEventData. The message is sent to the PublishSubscribeFrameworkExe1.
2. The PublishSubscribeFrameworkExe1 publishes anEvent and someEventData to the PublishSubscribeService using thePublishersProxyExe1.
3. The PublishSubscribeService looks for subscribers interested in anEvent. It finds one interested party and publishes anEvent with someEventData to thePublishersSkeletonExe2.
4. thePublishersSkeletonExe2 forwards the information to the PublishSubscribeFrameworkExe2.
5. the PublishSubscribeFrameworkExe2 forwards the information (via the same mechanisms as used in the previous scenario) to aSubscriberComponentExe2.

Specification

```
module PublishSubscribeService {


   exception IllegalPublicationSemantic
   {

   string reason;
   string action;

   };

   exception IllegalProperties
   {
   string reason;
   string action;

   };
   exception IllegalConstraints
   {

   string reason;
   string action;

   };

   exception IllegalFederation
   {

   string reason;
   string action;

   };
   exception BadEventData
   {

   string reason;
   string action;

   };
   exception IllegalSubscriptionFederation
   {

   string reason;
   string action;

   };

   exception ExpireTimeBeforePresentTime
   {
```

```
string reason;
string action;

};
exception CannotConnect
{

string reason;
string action;

};

exception SubscriptionDoesntExist
{

string reason;
string action;

};

typedef unsigned long SubscriptionHandle;
enum PublicationSemantics { ALL, ONE, BEST, ACTIVE };
typedef string Istring;
typedef Istring MachineId;
typedef Istring ExeId;
typedef Istring ComponentId;
typedef Istring ComponentInstanceId;
struct EventData
{

unsigned long publishTime;
MachineId machineId;
ExeId exeId;
ComponentId componentId;
ComponentInstanceId componentInstanceId;
any data;

};

interface Publisher
{

void publish( in PublicationSemantics publicationSemantics,

in CosTrading::Constraint constraints,
in CosNaming::Name nameServiceName,
in CosNaming::Name federation,
in EventData eventData)
raises( IllegalPublicationSemantic, IllegalConstraints,
IllegalFederation, BadEventData);

interface Subscriber
{
   // Expiration time: expressed as seconds since 00:00:00 January
   1,1901 UTC
   SubscriptionHandle subscribe( in CosTrader::PropertySeq properties,
   in CosNaming::Name nameServiceName,
   in CosNaming::Name federation,
   in unsigned long expireTime)
   raises( IllegalProperties, IllegalSubscriptionFederation,
   ExpireTimeBeforePresentTime);

   void unsubscribe( in SubscriptionHandle subscriptionhandle)

   raises( SubscriptionDoesntExist);
      };

   interface EventControl
   {

   void eventConnect(in CosNaming::Name nameServiceName)

   raises(CannotConnect);

   void eventClose(in CosNaming::Name nameServiceName);

   };
};
```

Related Frameworks

Framework: Component Interface Framework Overview

Framework: Trader Service Overview

Framework: Event Service Overview

Transaction Service Overview

Service Description

The Transaction Service provides transaction synchronization across the elements of a distributed component solution client/server application. Transactions are essential to the construction of reliable distributed solutions, especially those that require concurrent access to shared data.

The Transaction Service supports the concept of a transaction. A transaction is a unit of work that has the following (ACID) characteristics:

- A transaction is atomic; if interrupted by failure, all effects are undone (rolled back).
- A transaction produces consistent results; the effects of a transaction preserve invariant properties.
- A transaction is isolated; its intermediate states are not visible to other transactions. Transactions appear to execute serially, even if they are performed concurrently.
- A transaction is durable; the effects of a completed transaction are persistent; they are never lost (except in a catastrophic failure).

A transaction can be terminated in one of two ways: the transaction is either committed or rolled back. When a transaction is committed, all changes made by the associated requests are made permanent. When a transaction is rolled back, all changes made by the associated requests are undone.

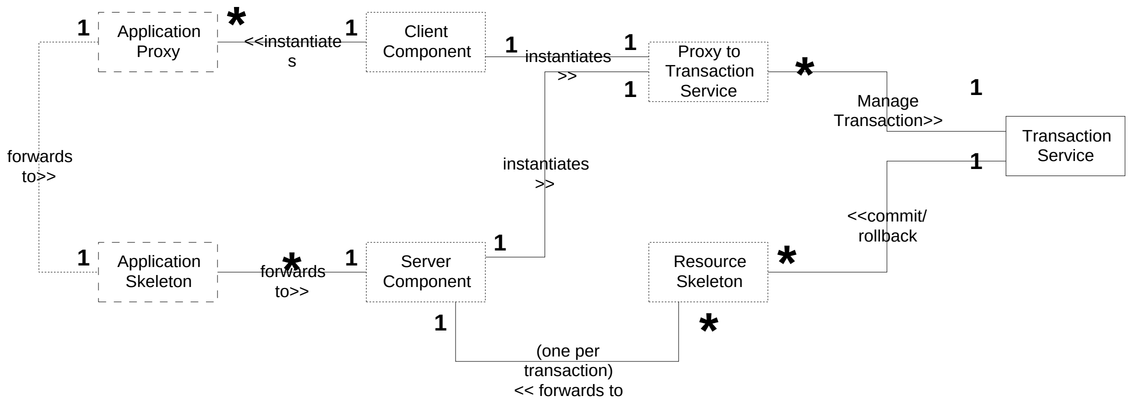
The Transaction Service defines interfaces that allow multiple, distributed components to cooperate to provide atomicity. These interfaces enable the components to either commit all changes together or to rollback all changes together, even in the presence of a failure. No requirements are placed on the components other than those defined by the Transaction Service interfaces.

Transaction semantics can be defined as part of any component that provides ACID properties.

A transaction can involve multiple components performing multiple requests. The scope of a transaction is defined by a transaction context that is shared by the participating components. The Transaction Service places no constraints on the number of objects involved, the topology of the application or the way in which the application is distributed across a network.

Structure

Class Diagram



Participants

Client Component

- Holds Proxies to application service interfaces
- Holds Proxy to Current interface on Transaction Service

Server Component

- Represents a component that provides ACID properties
- Encapsulates business logic
- Holds skeletons for all application services
- Holds Proxies to other components interfaces
- Holds Proxies to Transaction Service
- Holds skeleton to receive commands from the Transaction Service
- Interprets commands received from the Transaction Service

Proxy to the Transaction Service

- Transparently connects to the Transaction Service
- Used to begin, rollback, commit, etc. a transaction with the Transaction Service

Resource skeleton

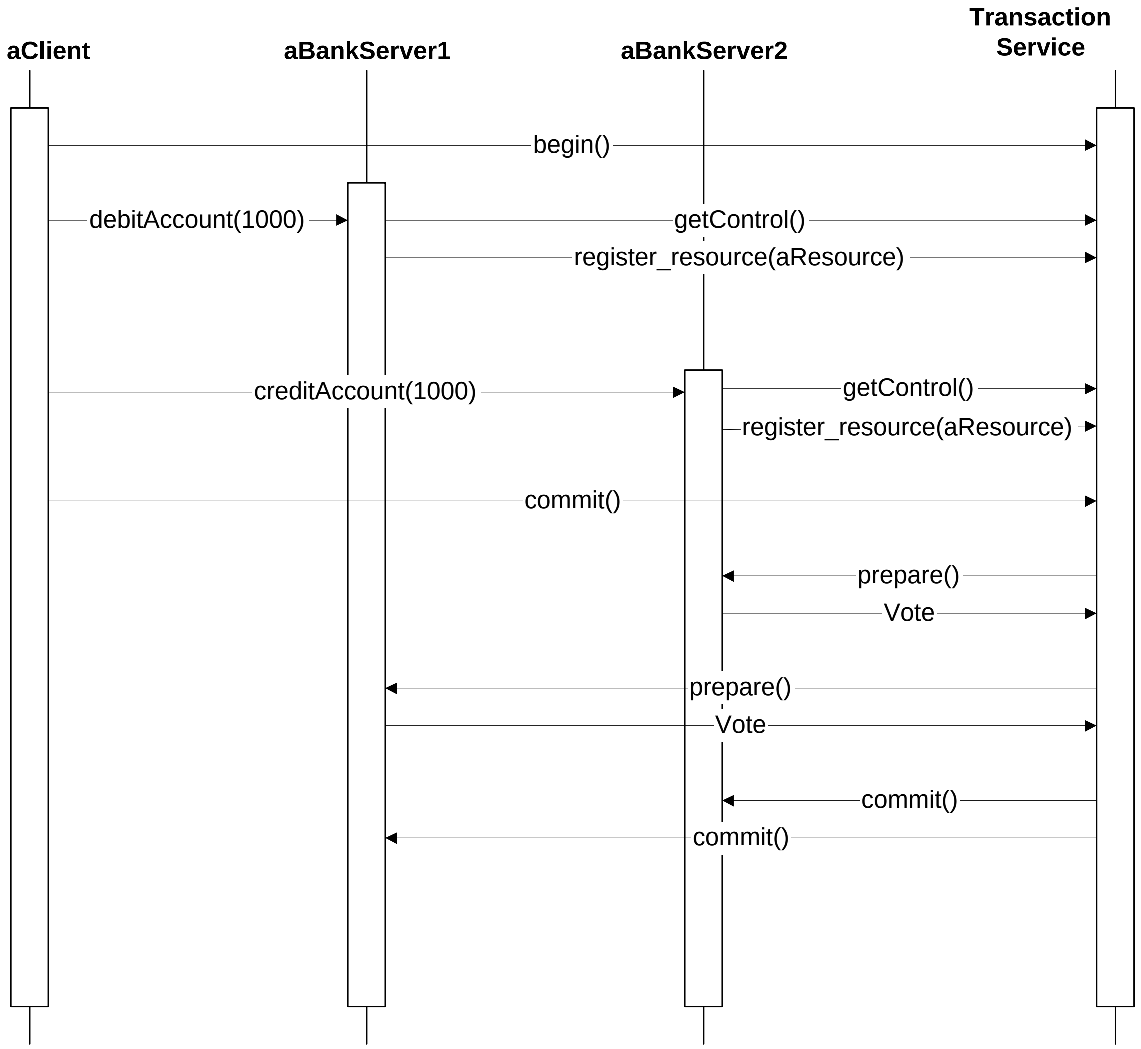
- Receives prepare, rollback, commit, etc. commands from the Transaction Service

- One per transaction

Collaborations

Scenario:

The following diagram shows the flow of requests to transfer $1000 between two different banking servers.



Stimuli:

Note: The transaction is implicitly sent by the middleware with every message.

1. aClient begins a transaction.
2. aClient debits an account on aBankServer1. aBankServer1 performs a getControl() from the Transaction Service and then a register_resource(). aBankServer1 does not debit the account at this time.
3. aClient credits an account on aBankServer2. ABankServer2 performs a getControl() from the Transaction Service and then a register_resource(). ABankServer2 does not credit the account at this time.
4. aClient sends a commit() to complete the transaction.
5. The Transaction Service sends a prepare() to each of the servers involved in the transaction (aBankServer1 and aBankServer2). The BankServers perform the first phase (of a two phase commit) and vote on whether or not to go forward with the transaction.
6. aBankServer1 and aBankServer2 vote yes so the Transaction Server signals each server to commit the transaction. The transaction is now complete.

Specification

Package: CORBA Service Specifications

Related Frameworks

Framework: Component Interface Framework Overview

Frameworks for Building Components

Introduction

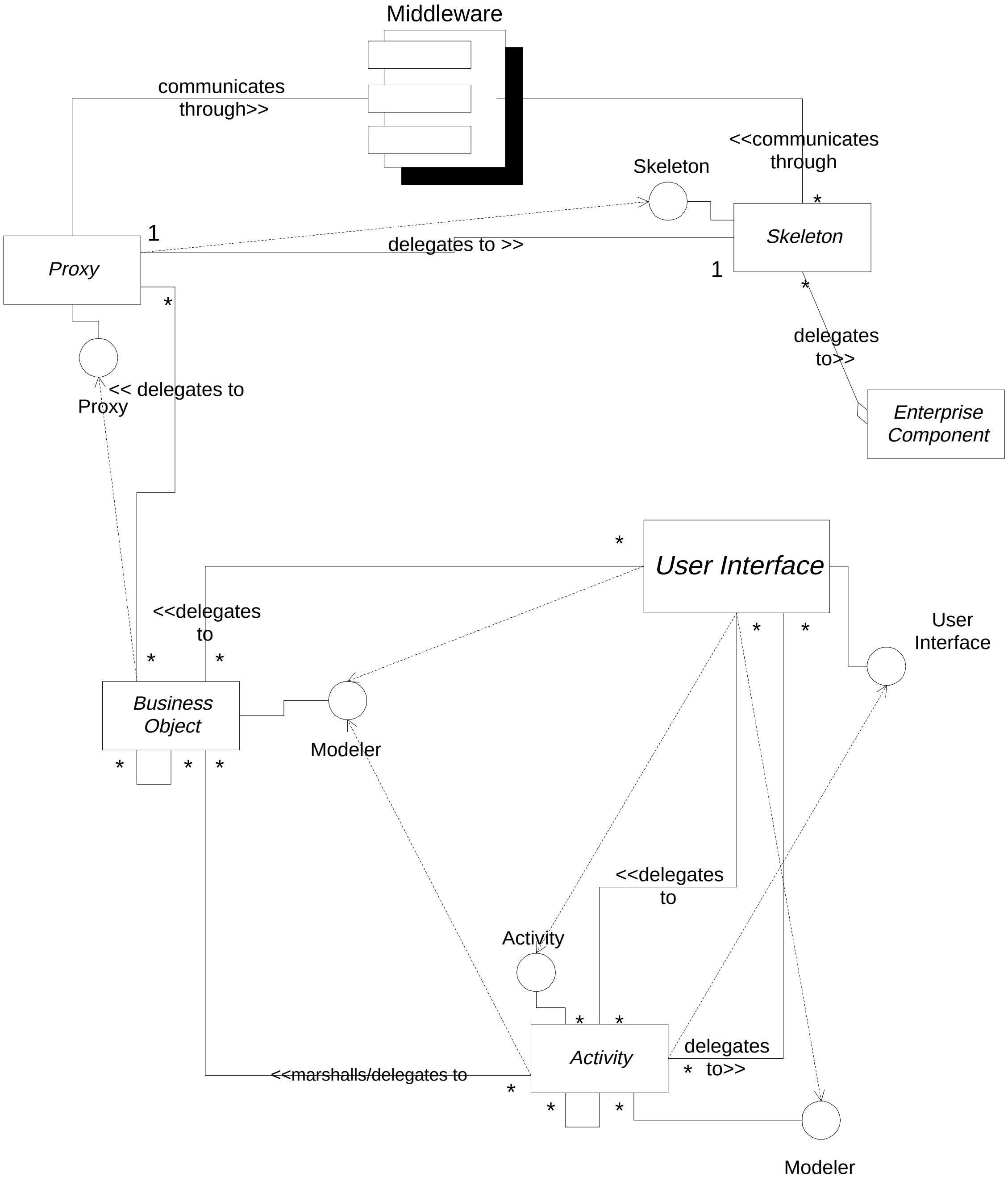
This framework category helps the developer build and introduce partitioning within a component. It is made up of several subcategories:

- User Interface Control
- Activity Control
- Business Class Modeling
- Business Class Persistence
- Component Interface
- Security
- IPS

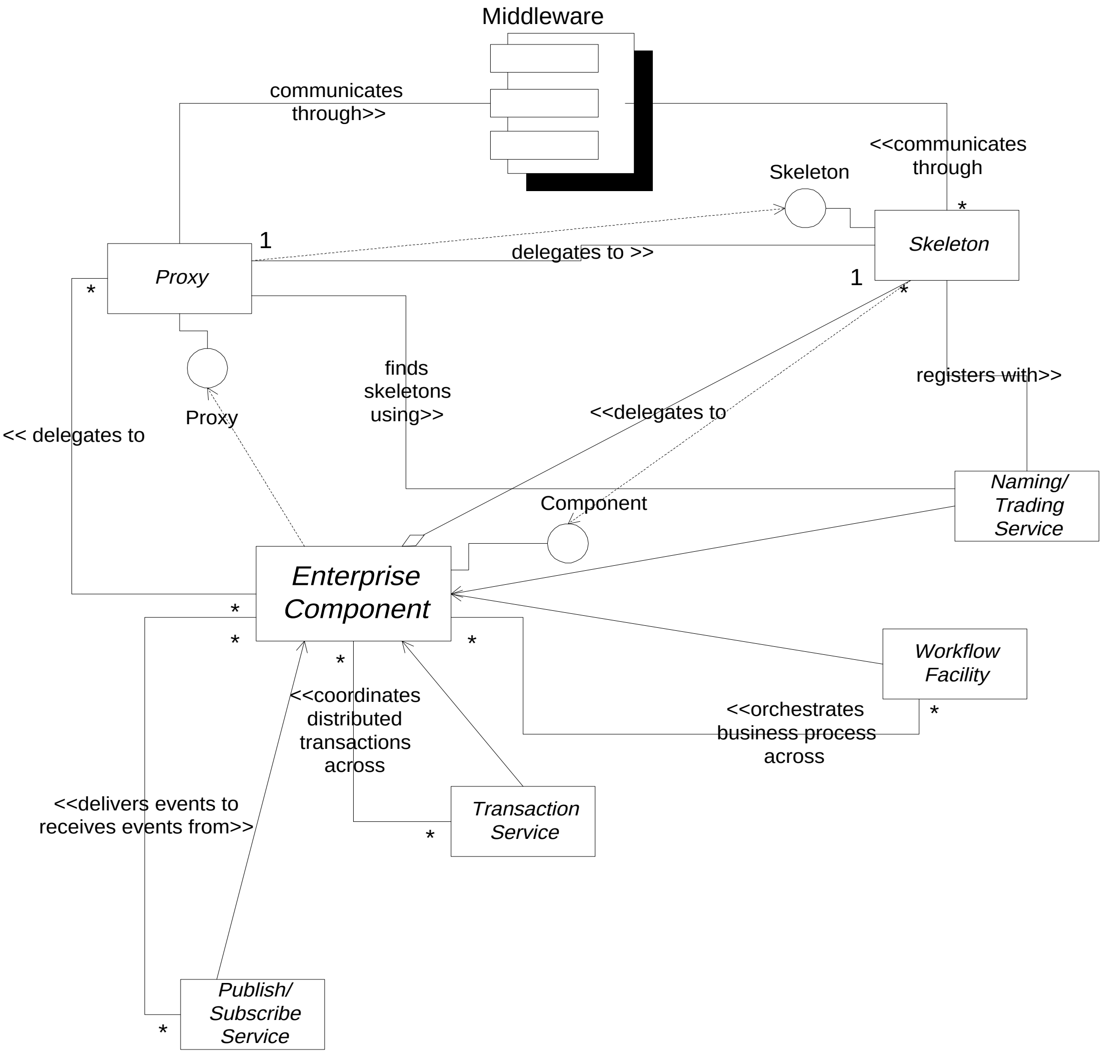
Metamodels

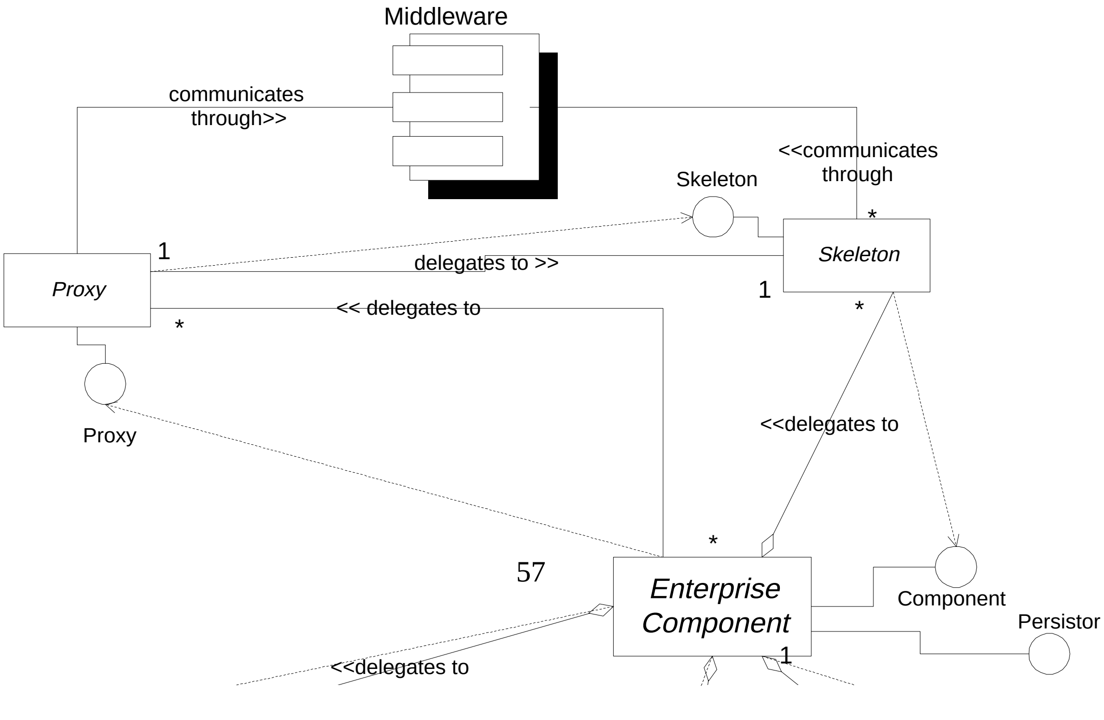
The following metamodels provide example implementations of the Frameworks for Building Components:

Client Application



Server Component





Integrated Performance Support (IPS) Overview

The IPS framework category supports users by providing information *about* the application through its subcategory frameworks:

- *Explain*. This framework facilitates the association between context-sensitive text and UI objects.
- *Task Assistance*. This framework facilitates the modeling, presentation, and enforcement of an activity's flow of control.

Explain Framework Overview

Purpose

The Integrated Performance Support (IPS) Explain Framework provides a flexible and configurable layer of IPS services. Explain is primarily a user interface concern. The overall purpose is to deliver predefined “explain” information to the user regarding some artifact (or widget) in the user interface (window, input field, button, etc.). The information can be presented in a number of ways. This specification presents one explain text presentation framework, the status text framework; however, the number of possible alternative presentation frameworks is limited only by a developer's imagination. New explain text presentation frameworks can be added without affecting the user interface. The explain information and the presentation of this to the user is separate from the application user interface itself, i.e. the user interface has no knowledge of IPS. This allows a plug and play approach; explain frameworks can be added or removed without affecting the application user interface. The explain framework supports the ability to add new explain text presentation frameworks as appropriate without modifying the application user interface.

Some possible explain text presentation frameworks are:

Status Text

> The status text framework displays a one line explanation in the status bar of the appropriate window whenever the mouse is over a widget that represents a public operation. The text is cleared when the mouse leaves the widget.

Bubble Help

> The bubble help framework displays a short text element in a small yellow box whenever the mouse pauses over a toolbar button. The text element is dismissed when the operation is invoked or the mouse leaves the toolbar button.

Popup

> The popup framework displays a long text element in a separate popup window whenever the mouse is over a widget. The framework could possibly be enabled by a user action, for instance clicking a predefined key. The popup window is dismissed by clicking or when the mouse leaves the widget.

Services

IPS Manager

The IPS Manager service provides a flexible and easily configurable way of setting up IPS functionality for user interfaces at run-time. Each time a user interface is opened, the IPS Manager determines what types of IPS are appropriate for this user interface and configures the user interface with the appropriate types of IPS. The parameters that contribute to determining the appropriate IPS may include (but are not limited to) the user interface, the user and application defaults (for example, for a particular application, the default may be that all users get status text information, but not bubble help information unless they activate it). The IPS Manager serves the purpose of clearly separating the IPS from the user interface itself. The advantage of using an IPS Manager is that the user interface does not know anything about what types of IPS are available and enabled. This allows IPS to be developed independently of the user interfaces, possibly by a different application team or even a third-party vendor.

IPS Router Lookup Strategy

The IPS Router Lookup Strategy service provides a flexible strategy for associating a new user interface (and its particular user) with a set of appropriate routers. The IPS Manager, which oversees this configuration, delegates the lookup of IPS routers to the IPS Router Lookup Strategy service. Delegating to a separate service allows different implementations to customize the IPS architecture with their own lookup mechanisms. This employs the well-known "strategy" pattern, as it encapsulates the lookup strategy, which is configurable, into a separate object. It clearly separates the application-specific lookup from the generic architecture methods.

The actual lookup can be accomplished in many ways. For example, this service could find matching IPS types in a keyed text file (perhaps in .INI format), relational or object database, hard-coded in an application-specific strategy (this would obviously not be reusable in other systems) or Windows Registry (on machines running the MS operating system).

Explain Router

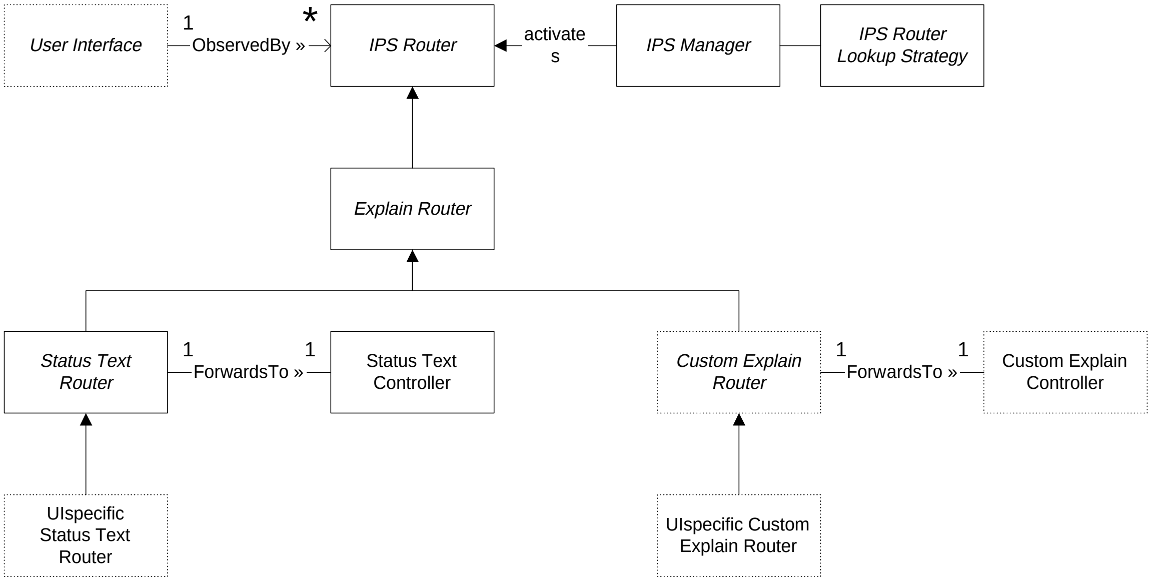
The Explain Router service provides a means of providing predefined explain information for the widgets in a user interface. The explain router offers methods for observing a user interface to look for specific events, and how to respond to them. The explain routers may consider user issues like skill level (e.g. beginner vs. expert), role (e.g. account manager vs. sales representative), language (e.g. French vs. English), etc. both in terms of what explain information to provide, when to provide it, and how to present it to the user. For instance, a beginner user may need help if he has been inactive for a little while, whereas an expert user probably needs help only when he requests it (by performing a user action). This specification presents one sample explain router, the status text router.

Explain Controller

The Explain Controller service provides a means of presenting explain information to the user. The explain controllers are created and controlled by the corresponding explain router. The explain controllers may have their own user interfaces, if appropriate, or they can affect the user interface that their explain routers are observing. This specification presents one sample explain controller, the status text controller.

Structure

Class Diagram



Participants

User Interface

- Declares an interface for being observed by the IPSRouters by implementing the Observable interface
- Broadcasts user interface events to all of its observers

IPSRouter

- Declares an interface for observing the User Interface by implementing the Observer interface
- Implements default behavior for setting itself up as an observer of the User Interface
- Implements default behavior for enabling and disabling its functionality
- Defines the common behavior of all IPSRouters

ExplainRouter

- Defines the common behavior of all ExplainRouters

StatusTextRouter (CustomExplainRouter)

- Defines the common behavior for all StatusTextRouters
- Defines what user interface events to respond to and how to respond to them
- Creates and manages the StatusTextController that will be used to present the status text information to the user
- Declares an interface for the mapping between widgets and status text information

UIspecificStatusTextRouter (UIspecificCustomExplainRouter)

- Implements the mapping between widgets and status text information for a particular user interface and user profile (skill level, role, language, etc.)

StatusTextController (CustomExplainController)

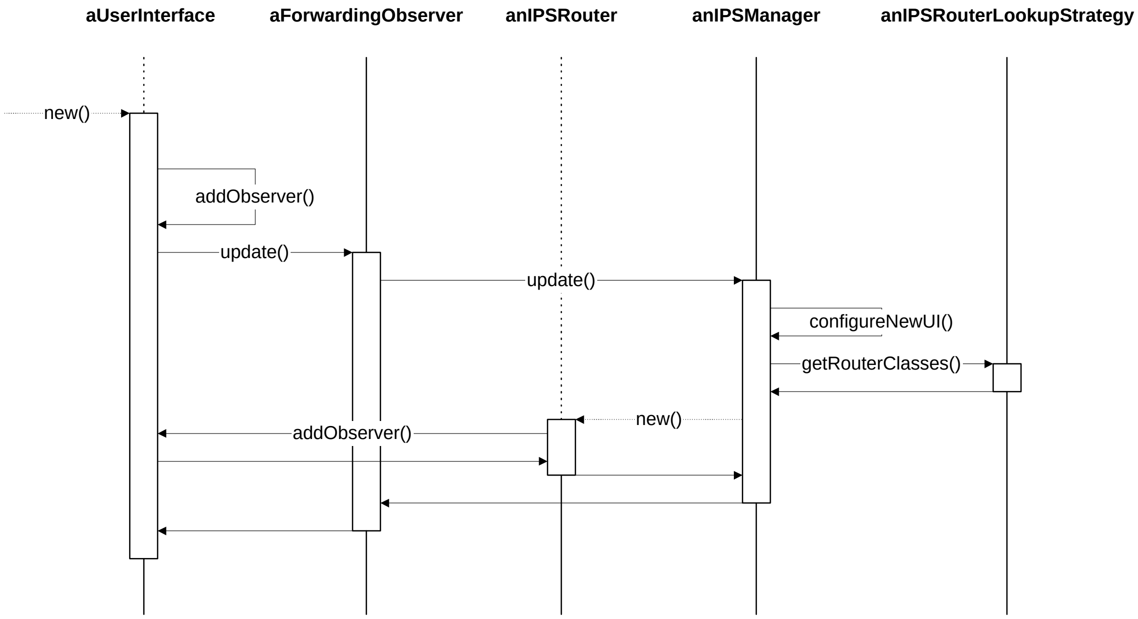
- Presents the explain information to the user
- Receives explain information from the CustomExplainRouter

Collaborations

Scenario 1:

Set up IPS routers when a new user interface is created.

The following diagram shows the flow of requests between the objects involved when a new user interface with one IPS router is created. It is assumed that the forwarding observer and the IPS manager are already created (this should be done when the application starts up). The IPS manager must be set up as an observer of the forwarding observer and must be initialized with an IPS router lookup strategy.



Stimuli:

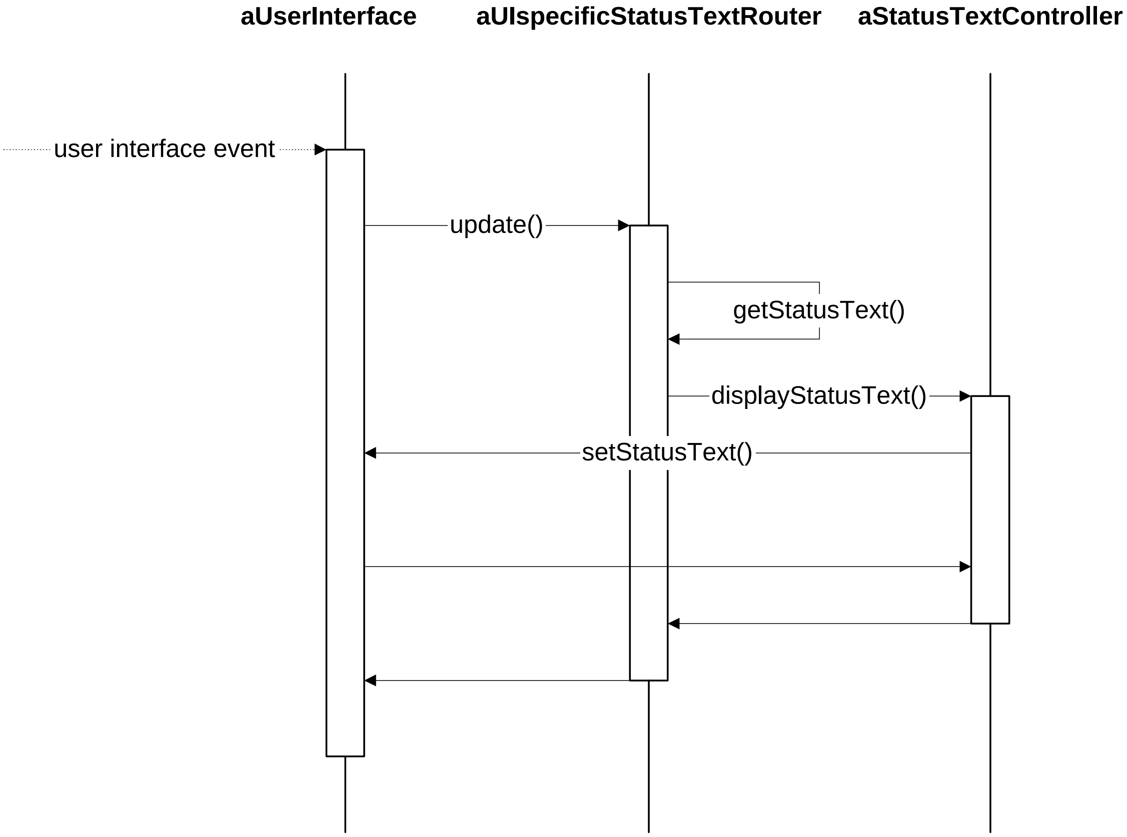
A new User Interface is created.

1. The User Interface invokes addObserver() to add the global accessible ForwardingObserver as an observer of the User Interface.
2. The User Interface invokes update() on each of it observers, i.e. the ForwardingObserver. The broadcast contains the User Interface itself and information that the provided User Interface has just been created.
3. The ForwardingObserver simply forwards the broadcast to all of its observers, in this case the IPSManager, by invoking update() on them. The ForwardingObserver preserves the broadcast in such a way that, for the observers, it looks like the broadcast came directly from the User Interface.
4. The IPSManager receives the broadcast, and based on that it is a notification of a user interface creation, it invokes configureNewUI() on itself. This is the method that deals with user interface creation. The IPSManager may receive other broadcasts, for example when the user interface is destroyed, and have to deal with them accordingly.
5. The IPSManager invokes getRouterClasses() on its associated IPSRouterLookupStrategy to delegate the lookup of IPS routers. The IPSRouterLookupStrategy applies its lookup mechanism, for example read from a file, to retrieve the IPS router classes this user interface should be configured with. The strategy returns the IPS router classes.
6. The IPSManager creates a new object of the IPS router class returned from the IPSRouterLookupStrategy. Any errors in creating the IPSRouter will be handled by the IPSRouterLookupStrategy.
7. The IPSRouter invokes addObserver() on the User Interface to add itself as an observer of the User Interface. By setting itself up as an observer, the IPSRouter will receive notifications when events, for example user interface events, occur in the User Interface (see Scenario 2).

Scenario 2:

Provide status text information when mouse moves over a widget.

The following diagram shows the flow of requests between the objects involved when the mouse cursor moves over a widget that has status text information associated with it. It is assumed that the status text router is set up as an observer of the user interface (see Scenario 1).



Stimuli:

A User Interface receives a user interface event from its native event handler.

1. The User Interface invokes update() on each of it observers, the IPSRouters, to broadcast that an event has happened. This broadcast contains at least the event type (MOUSE_ENTER) and the widget that produced the event.
2. The UIspecificStatusTextRouter invokes getStatusText() on itself to get the status text for the widget that produced the MOUSE_ENTER event. This returns the status text for the widget that produced the event.
3. The UIspecificStatusTextRouter invokes displayStatusText() on the StatusTextController to have the status text controller present the status text to the user.
4. The StatusTextController invokes setStatusText() on the User Interface to display the status text in the status bar of the appropriate user interface window. Since status text information is presented in the same user interface as the one being observed, the StatusTextController must request the User Interface of this user interface to display the status text in the status bar of this one.

Specification

Package: com.ac.eas.ips

Package: com.ac.eas.util

Package: com.ac.eas.userinterface

Related Frameworks

Framework: UIController

Task Assistance Framework Overview

Purpose

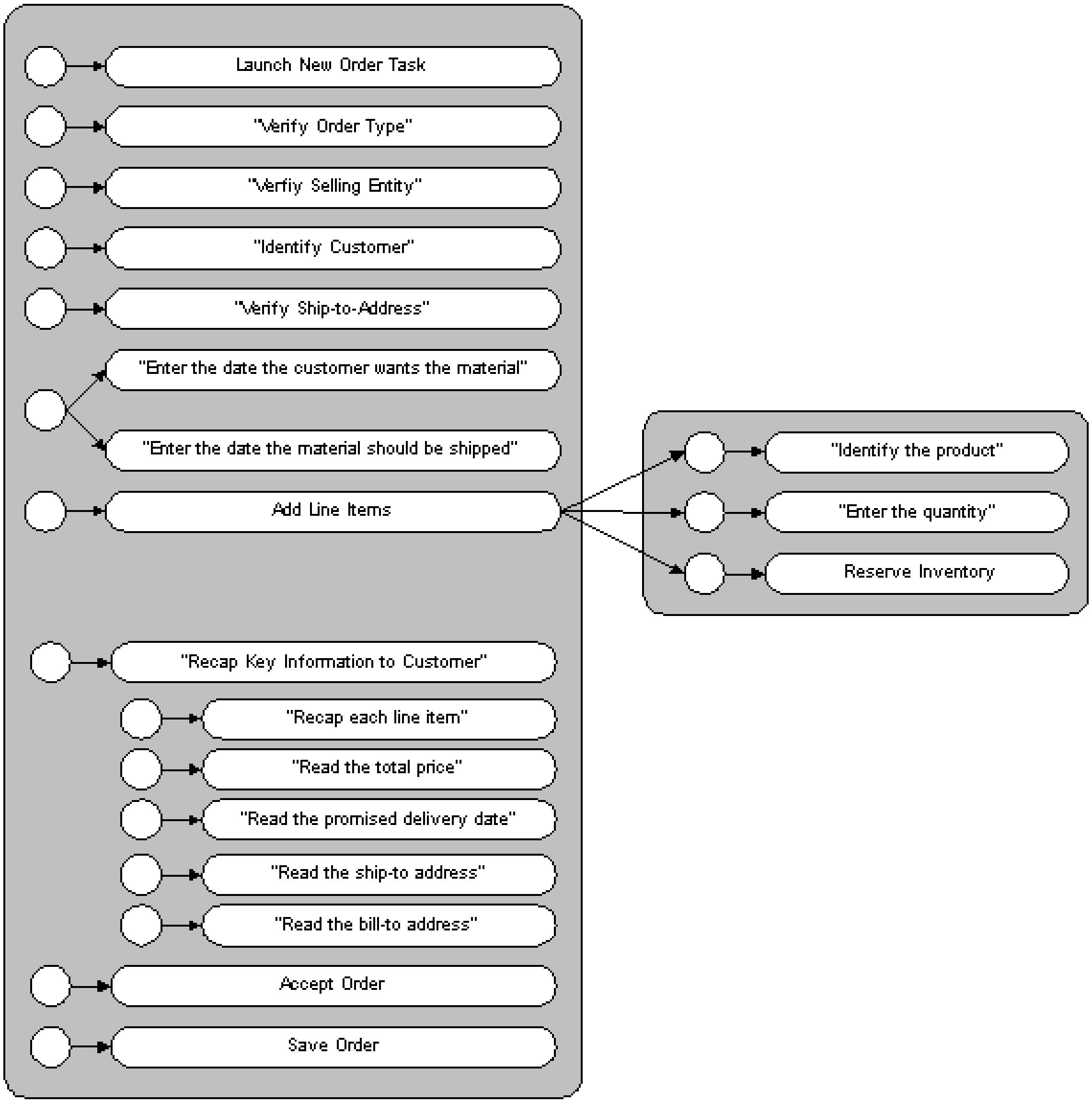
The objective of the Task Assistance Framework is to provide help to the User in completing the tasks assigned to him / her successfully.

The Task Assistance Framework provides the following capabilities:

- Allows organizations to specify "Scripts" to help Users accomplish their assigned tasks in a way that reflect their policies. A Script is a sequence of "Steps" with pre- and post-conditions that a User should perform. A Step can be an Activity, an operation of an Activity or business object, or a simple memo step, such as "Call Supervisor for Approval." The Step can only be executed if the pre-condition is passed (i.e. evaluates to true). A Step is considered "completed" only if its post-condition passes.
- Provides navigation control and visual assistance to the User in completing his / her assigned task. Task Assistance can include displaying information on the context of the Step (for example, purpose of the Step, whether the Step is required, how the Step fits into the Script, etc.) and stating the conditions under which the Step can / must be performed.

Example

The following diagram is an example of a Script for a User (most likely a Customer Service Representative (CSR)) to complete in entering an order:



The above Script models the series of Steps that a CSR must perform in order to ensure that an order is taken successfully.

A Script can be customized specifically for a particular kind of User. The Script for taking an order can thus vary dynamically depending upon the Role of the User performing the Script and the business policies set by the organization. For example, a mail-order company has a policy that requires that any order totaling more than $10000 must have a Supervisor approval. Therefore, the "Enter New Order Script" for a CSR Role would differ from a CSR Supervisor Role in that the CSR Script would contain an additional "Step" for orders totaling more than $10000 - the CSR must "Call the Supervisor for Approval".

To provide assistance to the User, a visual "task list" such as stated above can be built to provide navigation control as well as contextual information. Another option may be to build a "wizard" type of application to guide the User through each Step in the "Enter Order" Script.

Design Goals

The Task Assistance Framework provides a simple, flexible, and configurable layer that demonstrates the following characteristics:

- The Task Assistance Framework uses the Script as a model to provide visual assistance and navigation control to the User that is performing the Script.
- The Script model is used as a structure on which many different types of Task Assistance can be built. Thus a new Task Assistance Framework can be added without affecting the Script model.
- The Script description can be persisted in a number of forms (for example, flat file, relational database table, class with an instance variable holding onto a collection of Activities). It can be changed without changing the Task Assistance Framework that utilizes it.
- The Task Assistance Framework implements a clear separation of responsibilities between the visual representation of the Script model and the Task Assistance services. This allows transparent additions of new forms of Task Assistance (visual "task lists," CueCards, "wizards").
- The Task Assistance Framework is an optional extension to the User Interface Control Framework.

Services

Step

The Step service provides a way to represent an atomic task that the user has to perform. The Step may include information like:

- Display name (Enter Customer Name)
- State (COMPLETED, NOT_STARTED, IN_PROGRESS)
- Repeatable (Yes / No)
- Post-Condition
- Pre-Condition
- UI / Activity reference
- Step information

Script

The Script service is responsible for modeling the Script according to:

- A specific format (text file, object description, database table)
- The User 's Role (some additional steps must be performed by a user and not by a supervisor)
- The User's level of expertise (a Script for an expert user will not have the same level of details as a Script for a novice)

The Script service provides a way to define a sequence of Steps that need to be completed and to capture descriptive information about the Script such as its purpose.

The Script service defines a state for the Script that includes:

- Steps completed
- Steps to be completed
- Next Step to complete

The Script can be persisted by persisting the states of the Steps in the Script. The state of the Script is then derived from the state of the Steps.

Task Assistance

The Task Assistance service provides all of the basic behavior for implementing visual assistance to the User for completion of the Script. It offers the following services:

- Determine what type of Task Assistance layer (visual "task list," "wizard") is appropriate for the User executing the Script.

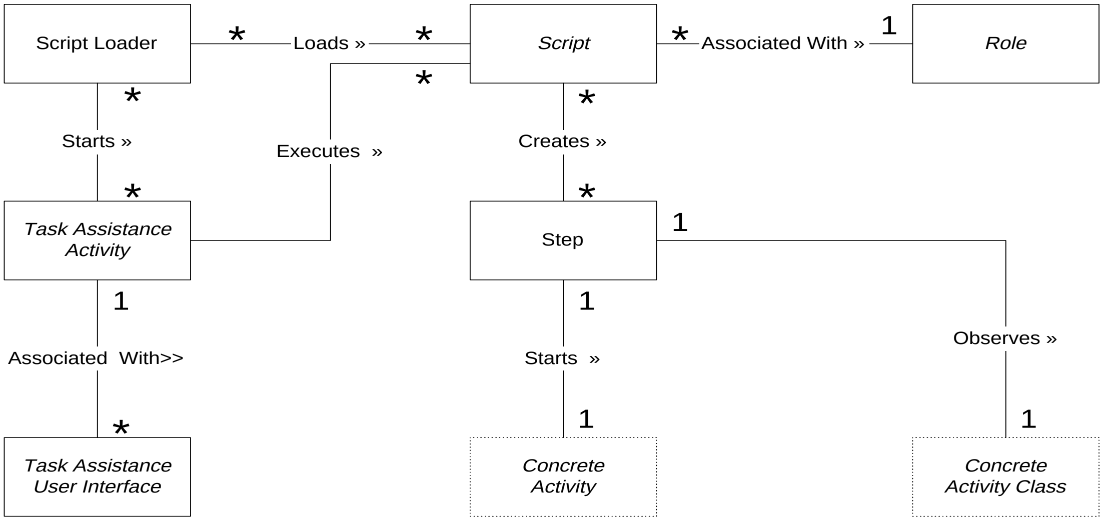
- Provide a visual representation of the Script.
- Display visual assistance to the User. Indicate which Steps have been performed and which is the current Step. Display contextual help for a specific Step.

The Task Assistance service is also responsible for:

- Controlling the execution of the Script by checking the state of the Step, the pre-condition and the post-condition.
- Invoking the Steps of the Script.
- Persisting the state of the Script.

Structure

Class Diagram



Participants

Script Loader

- Translates the Script persisted format into the Script Model internal format.
- Responsible for modeling the Script according to the level of expertise (e.g. beginner vs. expert), Role (e.g. Account Manager vs. Sales Representative) of the User.
- Responsible for selecting the appropriate Task Assistance layer (visual "task lists," "wizard") according to the User's preference and the application default.

Role

- Defines relevant information to construct the Script Model such as level of expertise, preferences, User's responsibility.

Script

- Represents the internal format for the Script Model.
- Offers interface for being persisted.

Step

- Represents an atomic action that needs to be performed.
- Contains information like name, state, pre-condition, post-condition, and Concrete Activity Class.
- Is able to start the Concrete Activity instance related to a Step.
- Observes the Concrete Activity Class to be notified of startup and end event occurring on the Concrete Activity instances.
- Is used by the Task Assistance Activity to invoke and monitor User's activities.
- Is used by the Task Assistance Activity to retrieve any help information for the Script execution.

Task Assistance User Interface

- Provides a visual representation of the Script model.
- Displays visual assistance to the User.
- Translates the user input event into a business event and forwards it to the Task Assistance Activity.

Task Assistance Activity

- Executes a Script by invoking the Steps and controlling the navigation between them.
- Retrieves help information for a Step.
- Persists the script.

Concrete Activity Class

- Declares an interface for being observed by the Step to allow creation of Activity instances independent of the Task Assistance layer.

Concrete Activity

- Started by the Step.
- Broadcasts startup and end messages to the Step.

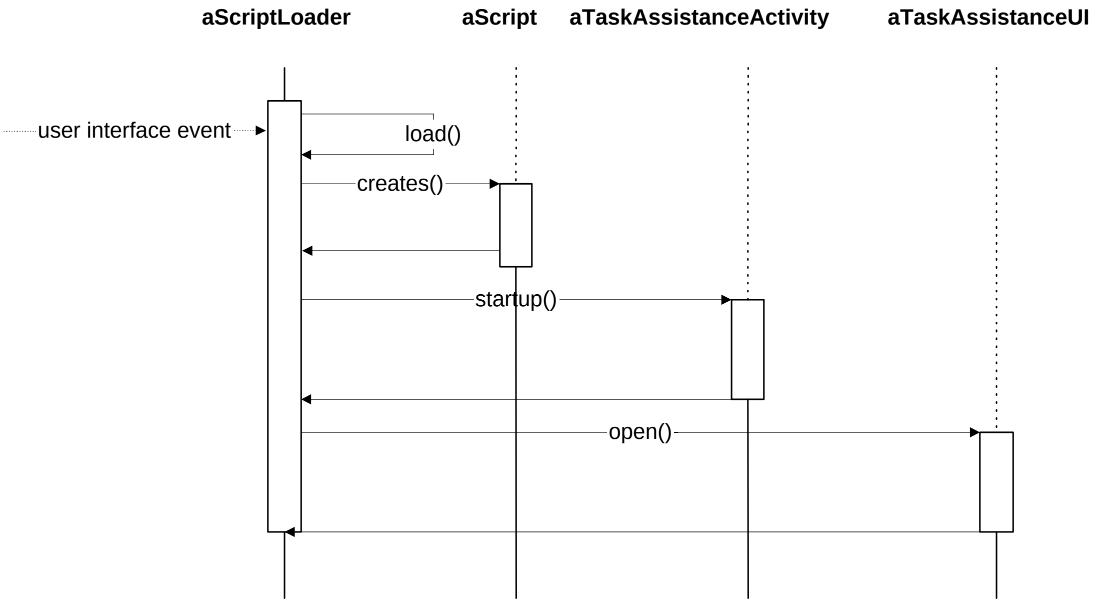
Collaborations

Scenario:

Start execution of a Script.

The following diagram shows the flow of requests between the objects involved when a new Script is loaded with the Script Loader.

Stimuli:



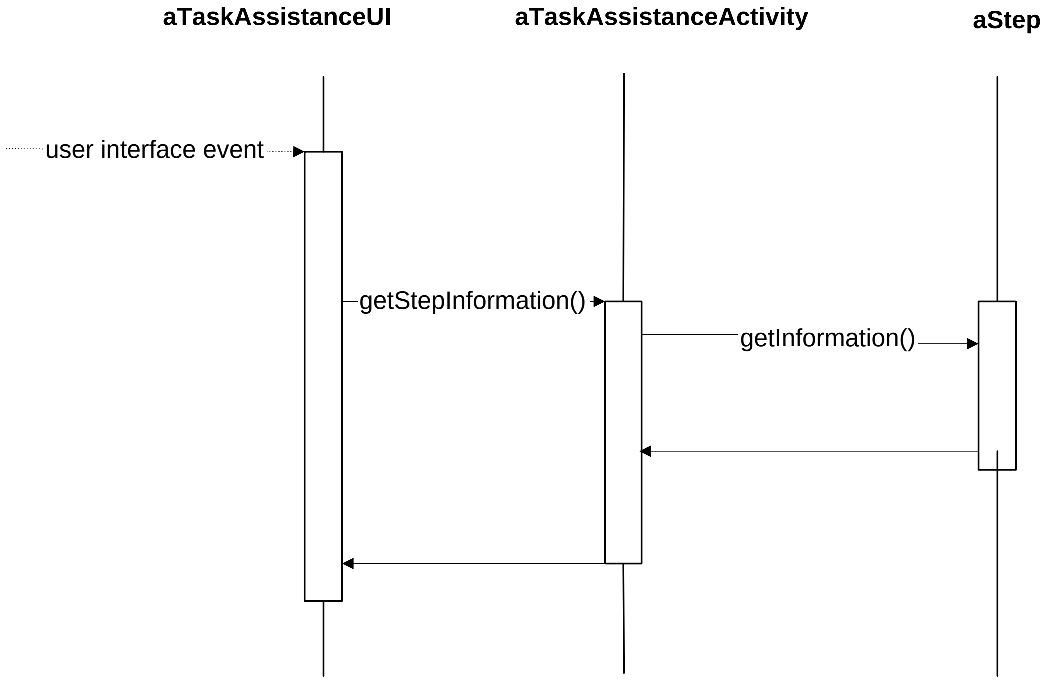
Script Loader receives a user event indicating that a new Script is started.

1. ScriptLoader invokes load() on itself to load the internal script format.
2. ScriptLoader invokes create() on Script to create the internal Script model.
3. ScriptLoader invokes startup() on TaskAssistanceActivity which executes the newly created Script.
4. ScriptLoader invokes open() on TaskAssistanceUserInterface to display the visual representation of the Script.

Scenario:

Provide Visual Assistance Information.

The following diagram shows the flow of requests between the objects involved when the User requests help for a Step of the Script.



Stimuli:

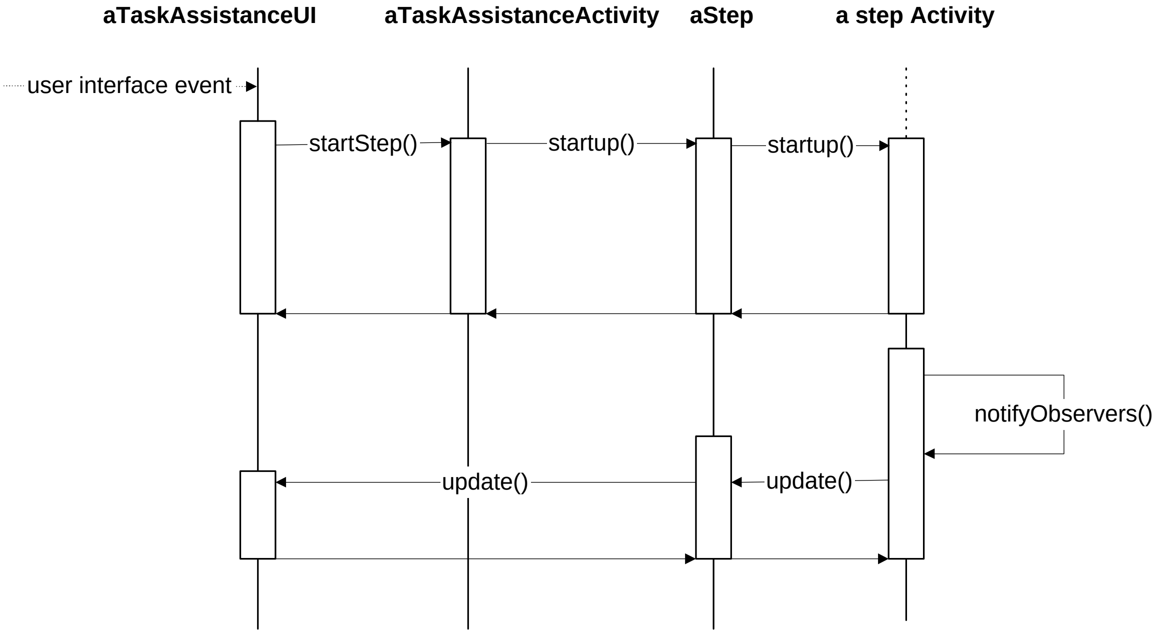
TaskAssistanceUserInterface receives a user interface event from its native event handler indicating that context information related to a specific Step needs to be displayed.

1. TaskAssistanceUserInterface invokes getStepInformation() on TaskAssistanceActivity to get information on a specific Step.
2. TaskAssistanceActivity invokes getInformation() on Step to get this information.

Scenario:

Start a new Step with the Task Assistance Layer.

The following diagram shows the flow of requests between the objects involved when the User starts a new Step of a Script.



Stimuli:

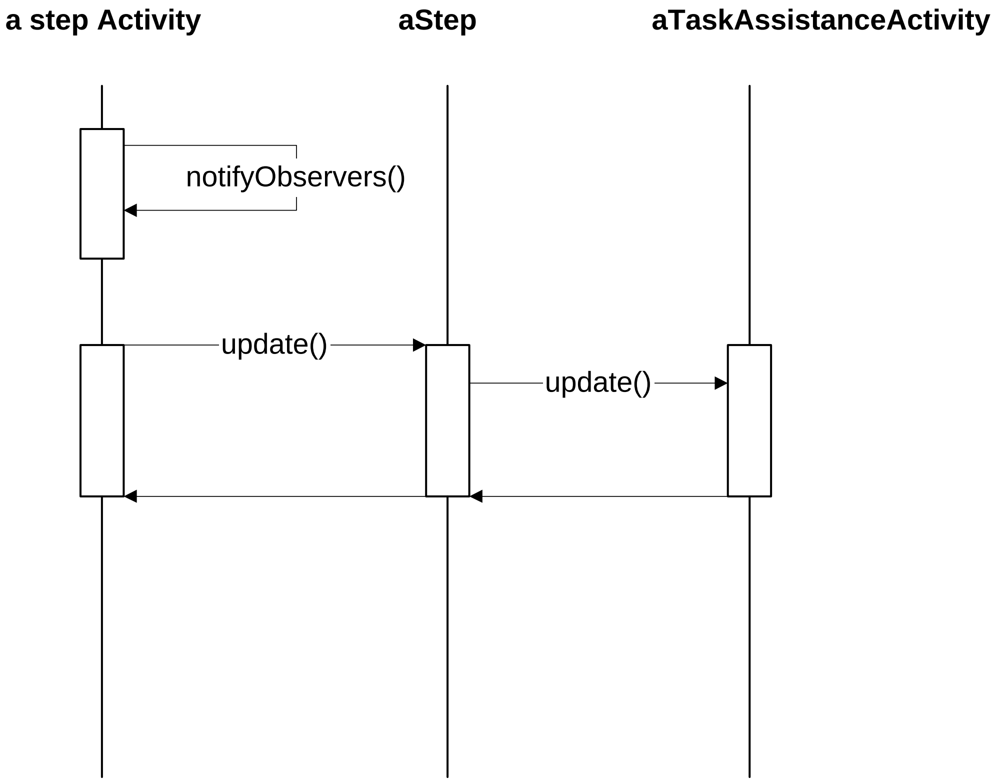
Task AssistanceUserInterface receives a user interface event from its native event handler indicating that a new Step must be started.

1. TaskAssistanceUserInterface invokes startStep() on TaskAssistanceActivity to start a new Step.
2. TaskAssistanceActivity invokes startup() on Step to start the Step.
3. Step invokes startup() on Activity to start the new Activity associated with the Step.
4. Activity invokes notifyObservers() on itself to broadcast the startup notification to all its observers.
5. Activity invokes update() method on each of its Observers.
6. Step forwards the update() method to the TaskAssistanceUserInterface to update its the visual representation of the Script execution.

Scenario:

End a Step.

The following diagram shows the flow of requests between the objects involved when a Step controlled by Task Assistance is completed.



Stimuli:

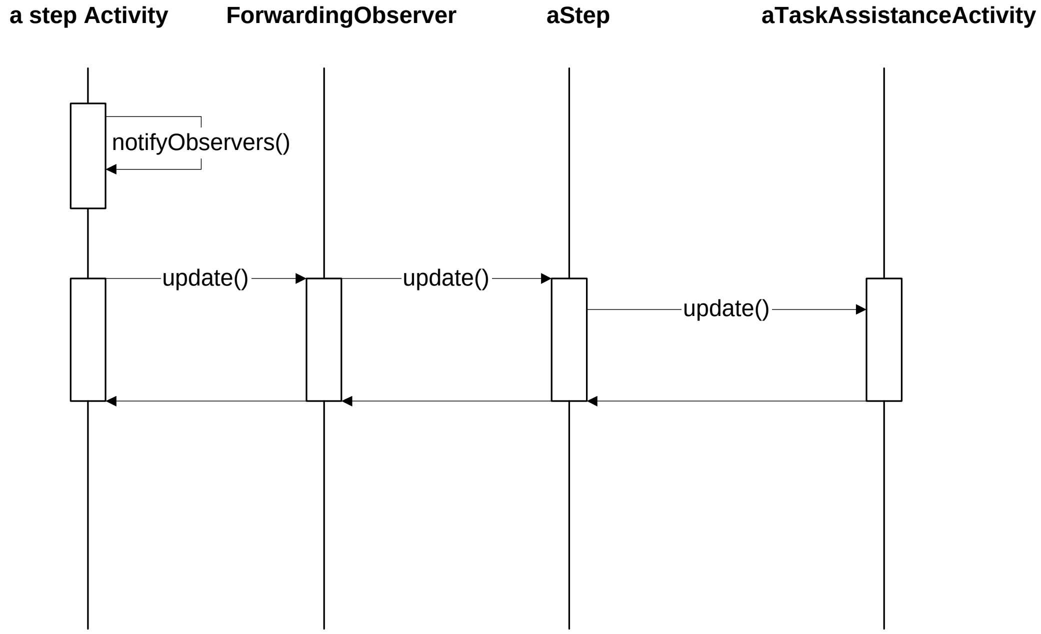
A step Activity receives a message indicating that it should stop.

1. Activity invokes notifyObservers() on itself to forward the end notification to all its Observers.
2. Activity invokes update() method on each of its Observers.
3. Step forwards the end notification to the TaskAssistanceActivity by invoking update() on TaskAssistanceActivity.

Scenario:

Start a new Step without the Task Assistance Layer.

The following diagram shows the flow of requests between the objects involved when a Step not controlled by Task Assistance starts.



Stimuli:

A new step Activity is started.

1. Activity invokes notifyObservers() on itself to forward the startup notification to all its observers.
2. Activity invokes update() method on each of its Observers.
3. ForwardingObserver is a class variable of the Activity Class. When a new instance of Activity is created, this class variable is set as an Observer of the newly created instance. Its role is to forward every broadcast message to the Objects that observe it and Step has been set as an Observer of the class variable ForwardingObserver of the Activity class. This allows Step to observe Activity instances before their creation.
4. Step forwards the startup notification to the TaskAssistanceActivity by invoking update() on TaskAssistanceActivity.

Specification

Package: com.ac.eas.activity

Package: com.ac.eas.util

Package: com.ac.eas.userinterface

Related Frameworks

Activity Framework

User Interface Framework Overview

Purpose

The User Interface (UI) framework provides services to model presentation look and feel and presentation-specific navigation. For example, a "maintain customer order" UI might need to support operations like "refresh order line list," "clear customer search fields," etc.

The User Interface provides separation of concern between presentation and business logic. The User Interface delegates all business-specific operations to Activity and Modeling objects. Separating the interface layer from the business logic increases reuse. Different types of user interfaces can reuse the same business logic. Likewise, the same interface can be used with varying business logic. Another goal of the separation of concern is to reduce the impact of change. For example, changing the user interface design should only impact user interface objects.

User Interface is the interface for classes that implement presentation layers. A developer implements this interface to create a solution-specific concrete UI class. A concrete UI models all of the operations that are needed to model presentation look and feel and presentation-specific navigation.

UIs are meant to implement UI-specific operations that leverage the more abstract business-specific operations of an associated Activity. The way UIs leverage Activities is by capturing UI specific events (specific button presses, menu selections, etc) and translating them into an invocation of an abstract business operation on the Activity. In addition many alternate UIs could be leveraging the same activity interface to perform the same basic business activity (maintain a customer order) but present a dramatically different look and feel (mouse/keyboard vs. touch screen, for example).

User Interfaces are meant to fill the role of "Interface" in the Interface-Control-Model Pattern.

Services

Status Text

User Interface provides basic status text functionality. The User Interface provides two APIs to set the status text (`clear` and `set`)

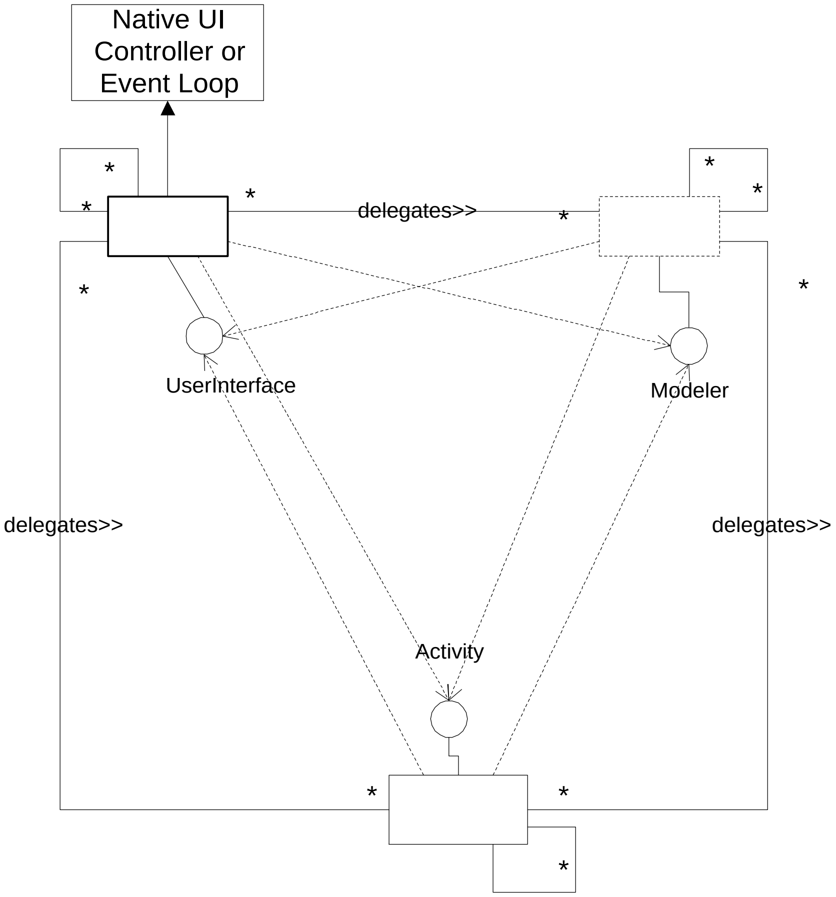
Open/Close

User Interface provides two APIs to `open` and `close` the user interface.

Note that other services would be added to specific user interface implementations to support the application specific features required.

Structure

Class Diagram



Participants

User Interface

- Interface implemented by objects that are responsible for presentation and UI event processing

Modeler

- Object which provides the modeler interface and represents a person, place, or concept of the business application

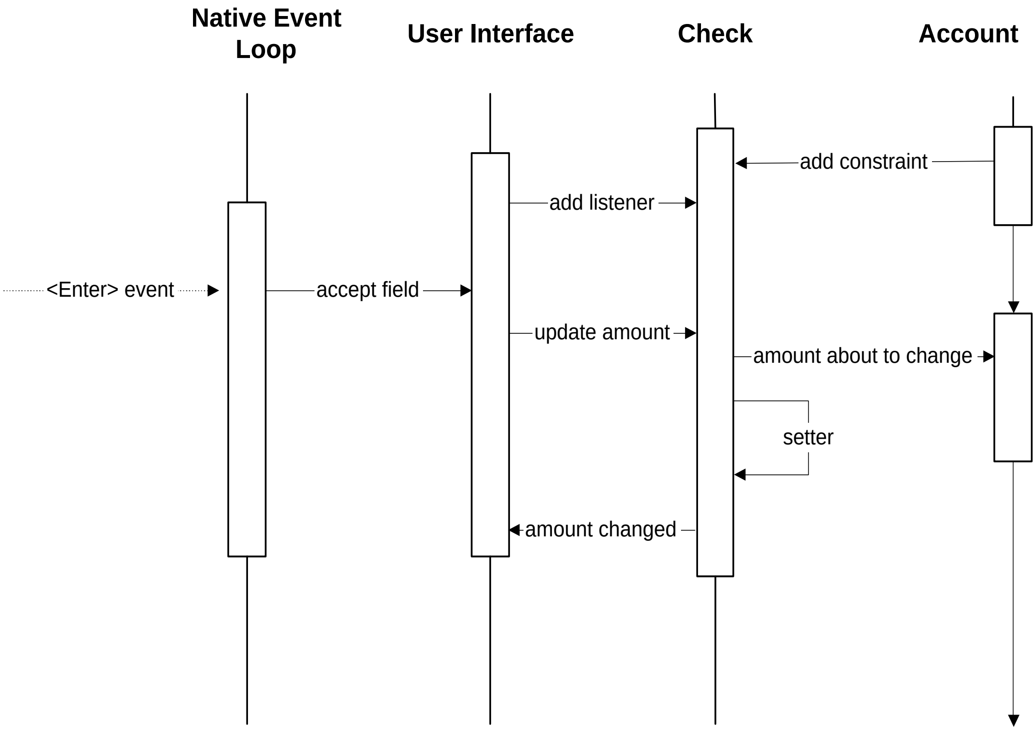
Activity

- Interface implemented by objects that model an application’s logical unit of work and provides “flow-of-control” logic

Collaborations

Scenario:

Update attribute of business object.



Stimuli:

1. Account adds itself as a constrainer on the check.
2. User Interface adds itself as a listener on the check.
3. User presses enter to accept text typed into field.
4. User Interface receives an accept field command from event loop.
5. User Interface sends an update amount to the check.
6. Within the setAmount method on the check, an amount about to change message is sent to all of its constrainers.
7. Since no constrainer including the Account, propagated an exception, the check updates its amount with the new value.
8. The check sends out an amount changed message to all of its listeners, including the User Interface.

Specification

Package: com.ac.eas. userinterface

Related Frameworks

Persistence Framework - Unit Of Work

Modeling

Activity Framework Overview

Purpose

The Activity framework provides services to model business units of work. The Activity, User Interface, and Modeling frameworks allow for the separation of concerns between user interface, business units of work, and model. The user interface represents the presentation and deals with user events. In contrast, the Activity represents the business logic completely isolated from any UI dependencies. Separating the user interface layer from the business logic allows different types of interfaces to reuse the same business logic. Likewise, the same user interface can be used with different business logic. Another goal of the separation of concern is to reduce the impact of change. For example, changing the activity’s flow of control logic should not affect user interface objects.

Activity is the interface implemented by objects which model business units of work. A developer implements this interface in order to create a solution-specific concrete business unit of work. A concrete activity class models all of the business operations that are needed to model the specific business activity.

Business activities are meant to implement reusable, abstract operations that could be used by other collaborating activities.

Activities are meant to fill the role of "Controller" in the Interface-Control-Model Pattern.

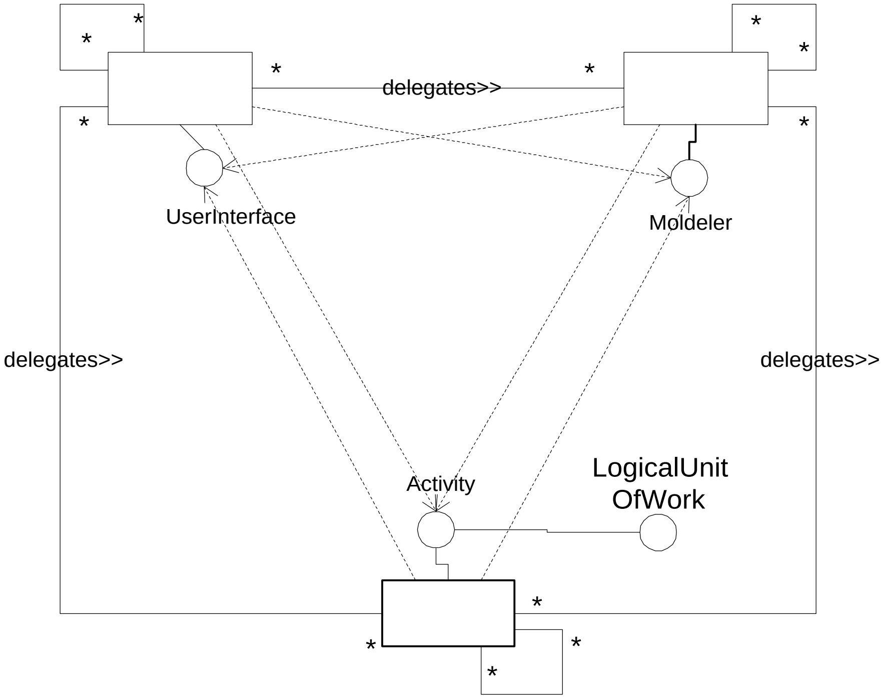
Services

Activity

The Activity service provides methods to control the life of the activity, such as startup and end. Activities are also responsible for managing its unit of work (commit, rollback, checkpoint, end) and its sub-activities.

Structure

Class Diagram



Participants

Activity

- Implements the common behavior for managing sub-activities and parent activity.
- The Activity interface should be implemented by concrete classes that are meant to implement reusable, abstract operations that could be used by other collaborating activities.

User Interface

- The User Interface interface should be implemented by classes that implement presentation look-and-feel and presentation-specific navigation. For example, it could be a window or an Applet.

Modeler

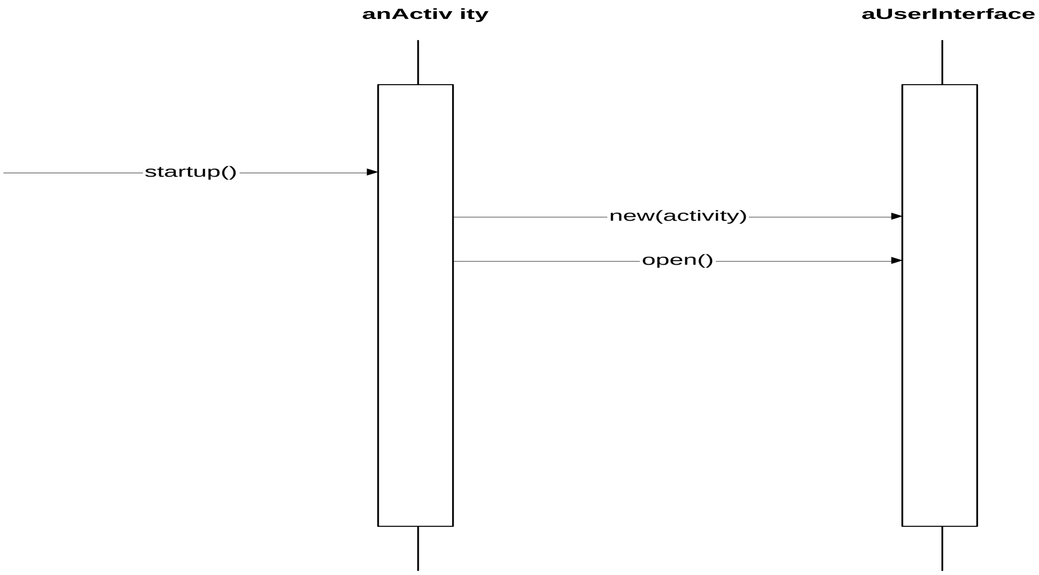
- Object which provides the modeler interface and represents a person, place or concept of the business application.

Collaborations

Scenario:

Start an Activity and its associated UI.

This is one pattern for starting a unit of work (which involves launching a user interface). In this pattern, the activity is the starting point. From the user's point of view, a user starts a unit of work to do his/her work. The unit of work (activity) may or may not launch a user interface to complete the unit of work.



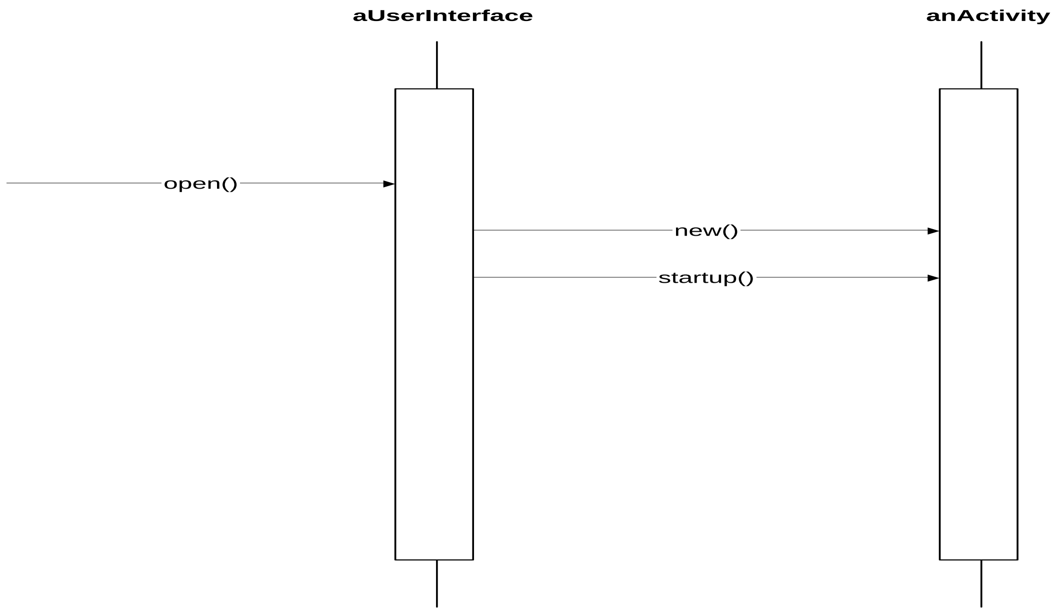
Stimuli:

1. The Activity receives the startup message.
2. The Activity creates a new User Interface, passing itself to the User Interface so that the User Interface can hold on to the activity and communicate with it. The Activity then opens User Interface.

Scenario:

Start a UI and its associated activity.

This is another pattern for starting a unit of work (which involves launching a user interface). In this case, a user would launch a user interface, which would then "attach" itself to an activity. The activity that the user interface "attaches" itself to can be a new activity that the user interface would startup or a currently "running" activity that the UI would invoke operations on.

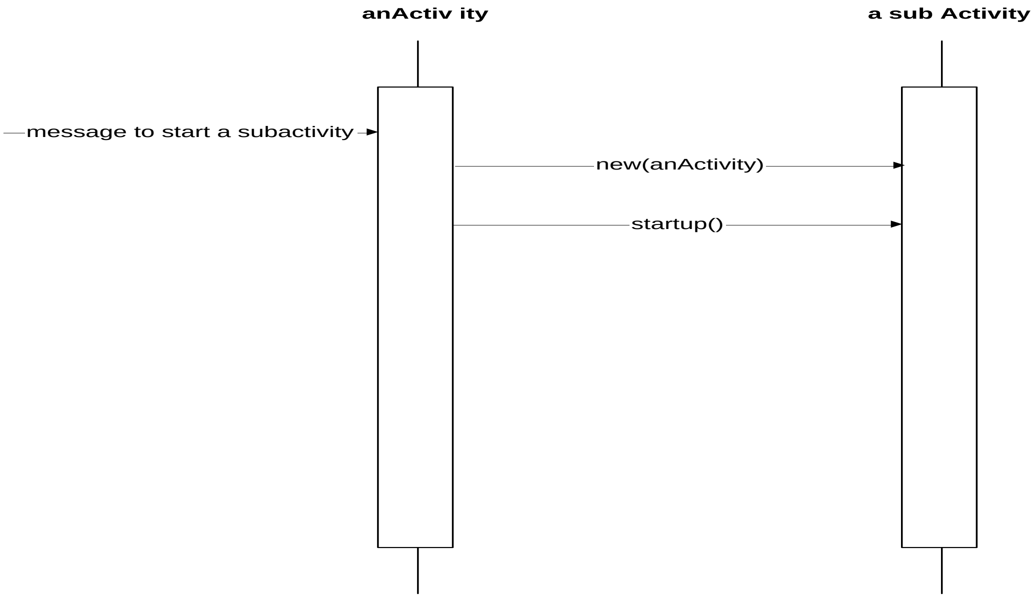


Stimuli:

1. The User Interface receives the open message.
2. The User Interface creates a new instance of its associated activity and starts the activity (which creates a new unit of work, does some initialization, etc.). The UI holds on to the activity to communicate with it.

Scenario:

Start a sub-activity.

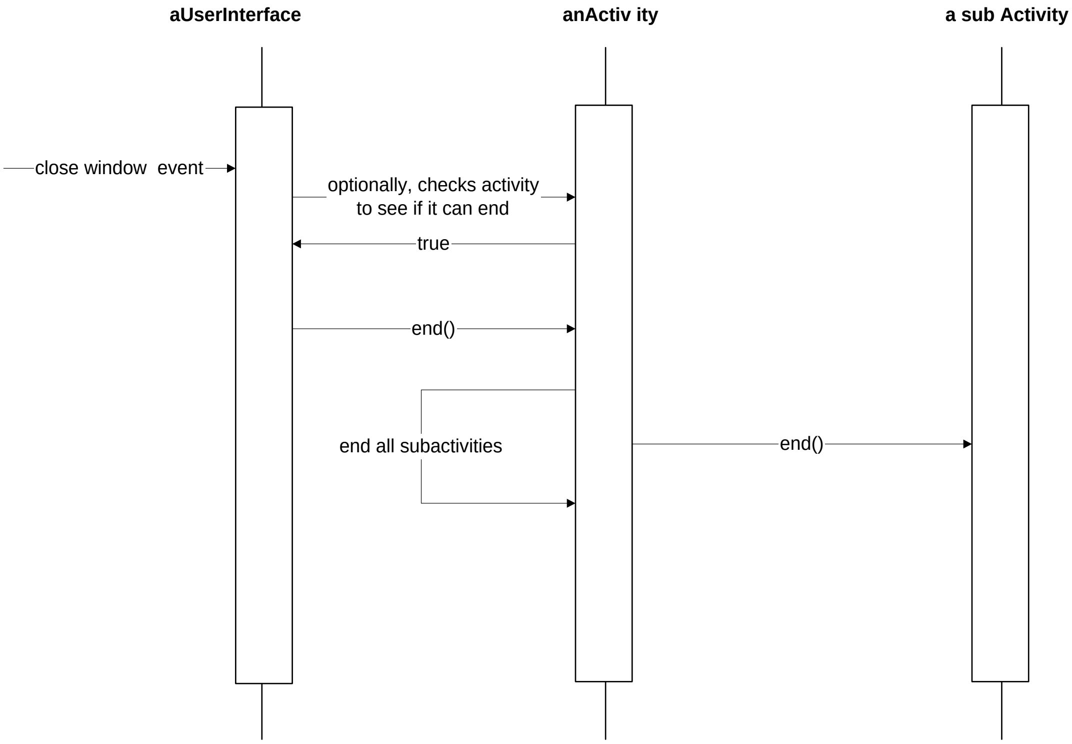


Stimuli:

1. An Activity receives a message to startup a subactivity.
2. The Activity creates a new instance of the sub Activity and passes itself to the subactivity so that the subactivity has a reference back to the parent activity.
3. An Activity has to somehow manage its subactivities (for example, having a collection of subactivities as an instance variable or having each subactivity as an instance variable).
4. The Activity then startups the subactivity by sending startup() to it. The subactivity then proceeds with its own processing.

Scenario:

Ending an activity and its sub-activities.



Stimuli:

1. The User Interface receives a close window event. The User Interface may invoke some method on its activity to check if the activity has finished its work and if the User Interface can close.
2. If the activity can end, then the User Interface invokes `end()` on its activity to end the unit of work. The User Interface then executes its closing window logic.
3. The Activity then cycles through its collection of sub-activities and invokes the `end()` method on each of its sub-activities.
4. The Activity then finishes its end activity logic.

Specification

Package: com.ac.eas.activity

Related Frameworks

Framework: Integrated Performance Support

Framework: User Interface

Framework: Business Object Modeling

Modeling Framework Overview

Purpose

A number of modeling services are provided here which aid in the defining and describing of an object's publicly exposed attributes, operations, and events. A definition and description of an object or model formally defines some entity within the business component (activities, business objects, or imported component interfaces) and its relationship with other objects. A model can provide meta-data related to change notification, constraints, editing parameters, typing, and policies that can be used at both design and run-time. Additionally these modeling services provide the APIs needed for other objects to access the meta-information. These services also provide support for establishing bounding and constraining relationships between objects.

Services

Constraints

Using the constrained and constrainer interfaces, objects can have constraints placed on their state. A business object can have its overall state constrained. Validity of the object can be based upon the values of its attributes.

Using this framework, Classes can provide:

- Attribute validation - Variables of an object can be constrained by an object implementing the constrainer interface (an attribute object).
- Operation Enablement - The availability of an operation can be constrained by an object implementing the constrainer interface (an operation object).

Run-time Meta-data Services

The meta-data service is made up of interfaces which allow objects to provide useful information about their members (attributes, operations, and events) and the common interaction patterns between these members.

Displayable

- Member display strings
- Names
- Labels
- Short Description text

Type

- Types, including base types as well as complex types (monetary, business type)
- Default values

Policy

- Derivation value policies
- Validation policies
- Operation availability policies

Editing

The editing service allows objects to provide useful editing information about their attributes / operation / events.

- Display length
- Display format

The editing service also allows objects to provide editable attributes. By acting as a mediator between the model and GUI controls, these editable attributes can provide typing validation, prescribed list validation, and conversions to and from strings.

Interaction Modeling

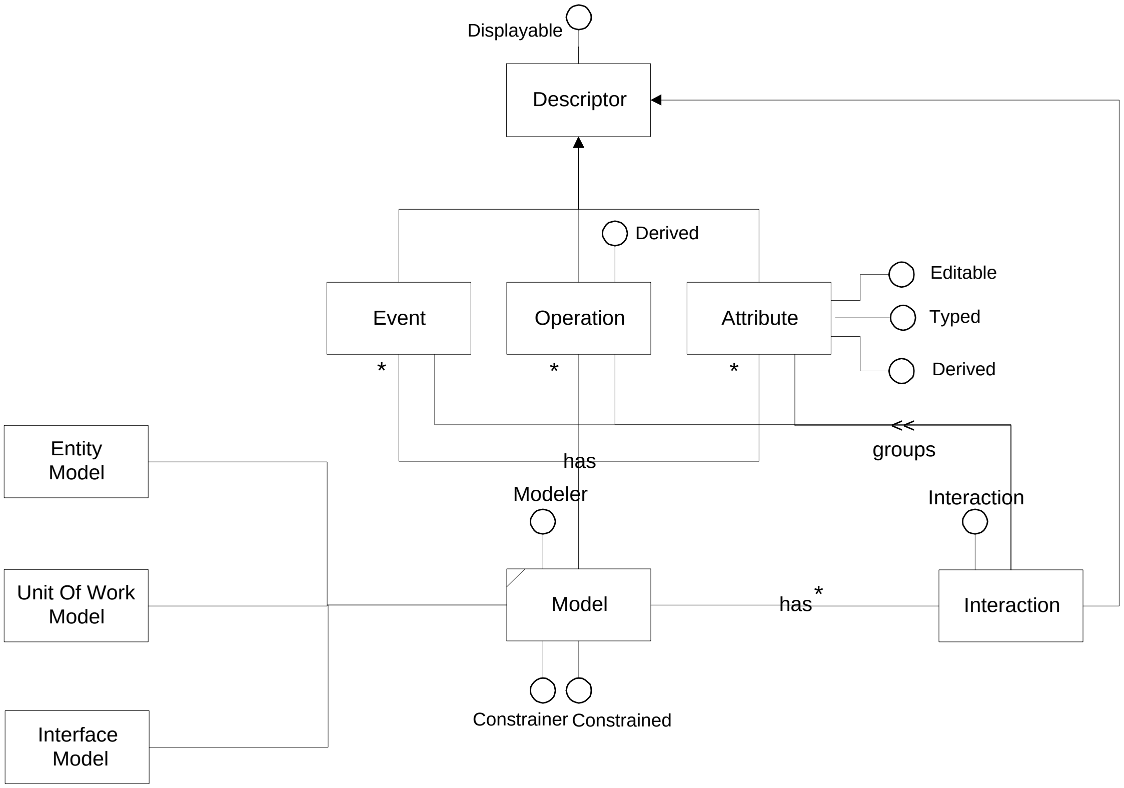
The modeling information should be packaged in a way which facilitates common interaction patterns for the business object. One example of an interaction model would be “enter order” for an order. Creation of an order involves entering header information and order lines, followed by a call to “place order.” “Enter order” should not provide certain information such as shipment IDs, order line unit costs, or other information which is not pertinent to creating of an order. This interaction model would be much different than one which provided a searching interface on the object. In this case the order may only provide those attributes which can be used for searching.

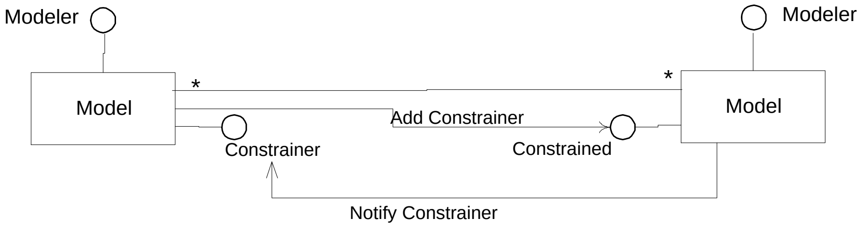
An interaction model consists of a collection of attributes, operations, and events. An interaction model can be used to describe the inter-dependencies of attributes, operations, and events.

- Attribute Dependencies - An attribute’s value is recalculated when another attribute changes, an operation is invoked, or an event is fired. For example , interaction descriptors can be used to describe the attributes which are involved in the calculation of a derived attribute.
- Operation Enablement State - An operation’s enablement state is tested when another operation is invoked, an attribute changes, or an event is fired. For example, interaction descriptors can be used to describe the attributes which are required before an operation can be invoked.
- Event triggering - An event is triggered whenever an attribute changes or an operation is invoked

Structure

Class Diagram





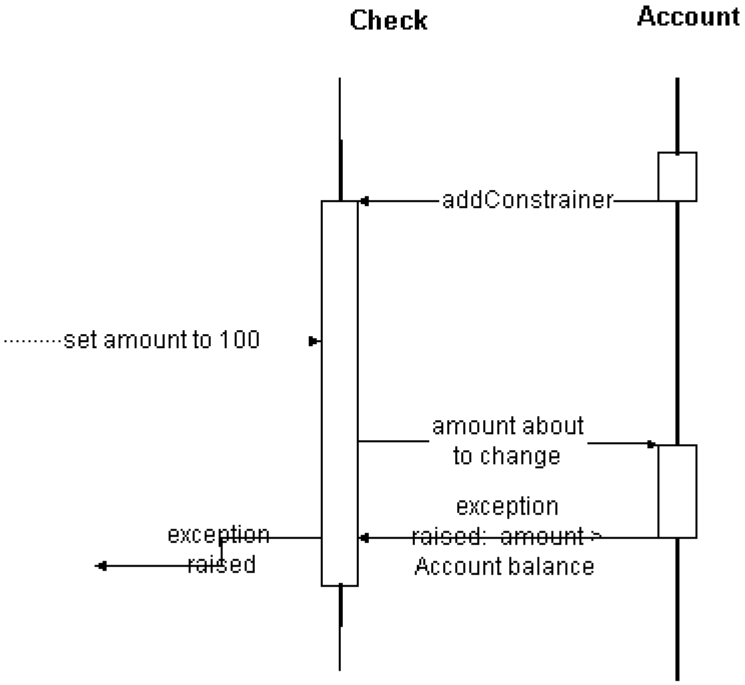
Participants

- *Constrained* - Interface which specializes the bounded interface and provides the ability for the implementer to have constraints placed on it. A constrained object can keep a list of it's constrainers and an API for adding/removing Constrainers. Constrainers, when asked, can evaluate a constraint with the constrained object as the subject.
- *Constrainer* - Interface for implementers which enforce constraints on a Constrained object.
- *Displayable* - Interface that can be implemented to provide display characteristics.
- *Editable* - Interface that can be implemented to provide editable characteristics.
- *Typed* - Interface that can be implemented to provide type and default values.
- *Policy* - Interface that can be implemented to provide policies for a member (derivation, validation, availablity, etc.).
- *Modeler* - Interface which provides an object with the capability of having members (attributes, operations, and events).
- *Model* - Concrete object representing one of three different types of models.
- *Interface Model* - Object which provides the modeler interface and wraps a proxy to a component interface.
- *Business Object* - Object which provides the modeler interface and represents a person, place or concept of the business application.
- *Unit of Work Model* - Object which provides the modeler interface and represents a business unit of work.
- *Attribute* - Object describing an exposed public variable of a model, represented by a getter and a setter method.
- *Operation* - Object describing an exposed public behavior of a model.
- *Event* - Object describing an exposed public event, that a model can fire.
- *Interaction* - Grouping of exposed public attributes, operations, and events.

Collaborations

Scenario:

Update attribute to invalid value.

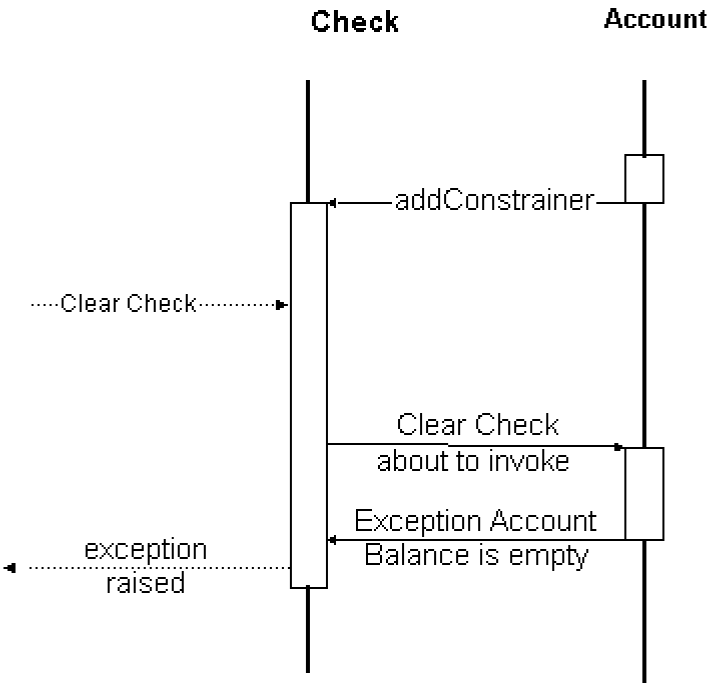


Stimuli:

1. Account adds himself as a constrainer on the Check.
2. Check receives a set amount to 100.
3. Within the setter method, the check broadcasts an amount about to change message to all of its constrainers, passing along the attribute name, the old amount value, and the new amount value.
4. Account evaluates its constraint for the check, and the constraint fails and an exception is raised.
5. Check object does not update the amount and propagates a validation failed exception.

Scenario:

Invoke unavailable operation (Fire invariants).

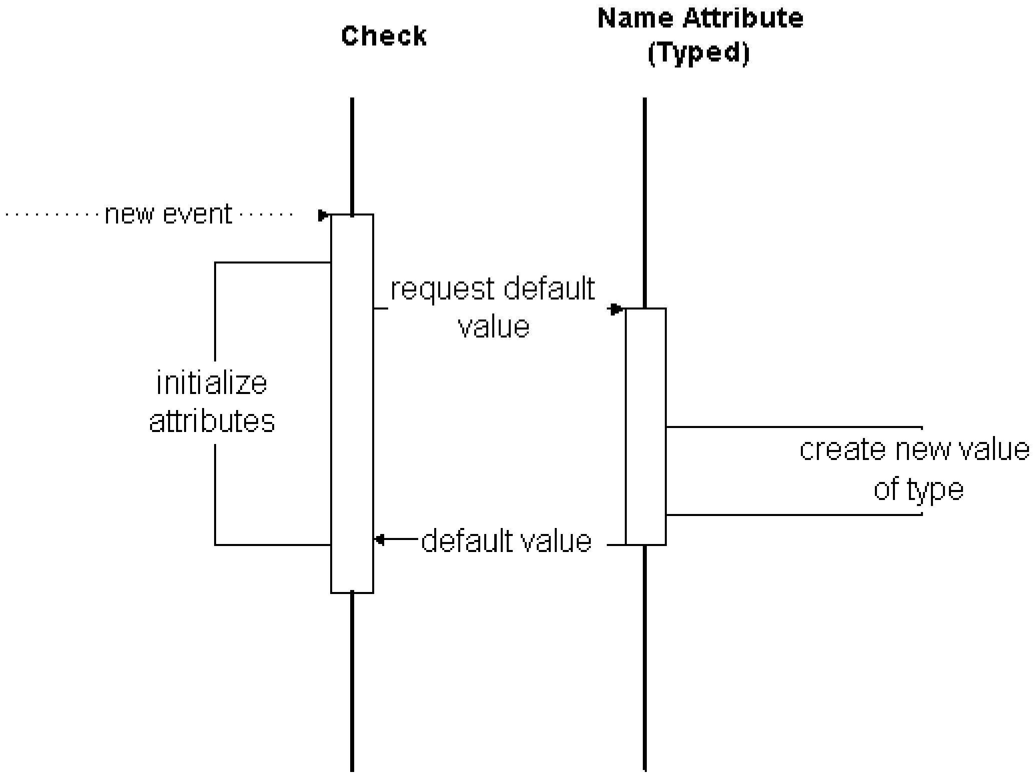


Stimuli:

1. Account adds itself as a constrainer on the check.
2. Check receives a clear check message.
3. Check broadcasts an operation about to invoke message to all of its constrainers, passing along the operation name.
4. Account evaluates its constraint for the check and the constraint fails and an exception is raised.
5. Check does not invoke the operation and propagates a operation invoke failed exception.

Scenario:

Static default value.

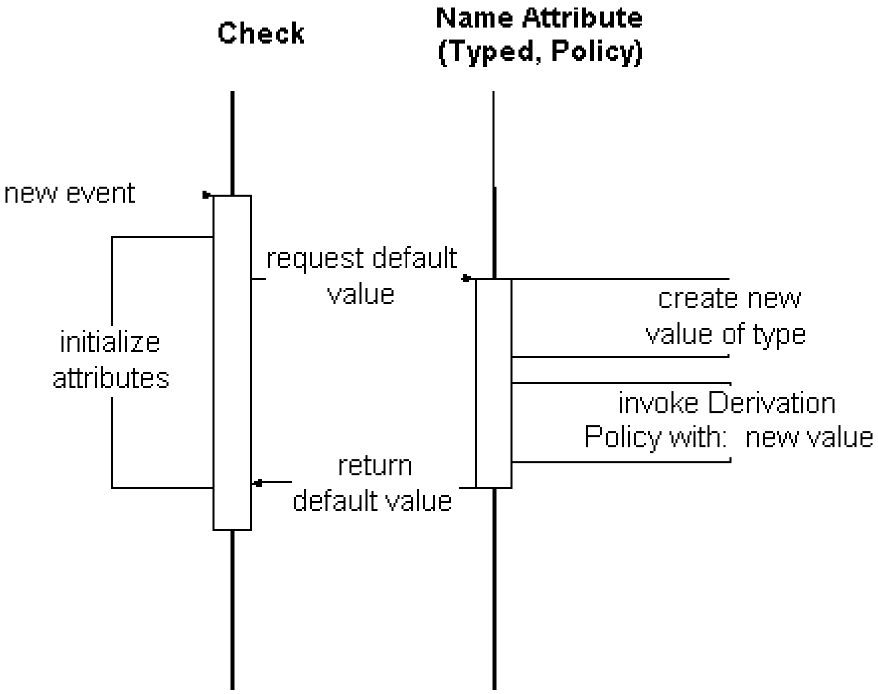


Stimuli:

1. Check receives an initialize method.
2. Check asks the name attribute for the default value.
3. Name attribute creates a new value of its attribute type.
4. Name attribute returns the new value.

Scenario:

Derived default value.

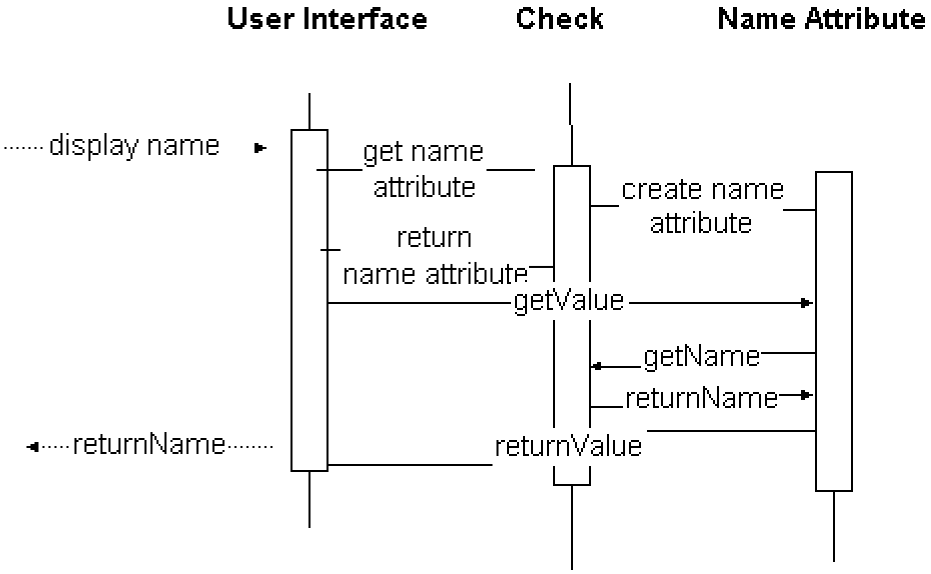


Stimuli:

1. Check receives an initialize method.
2. Check asks the name attribute for the default value.
3. Name attribute creates a new value of its attribute type.
4. Name attribute executes its derivation policy method on the new value.
5. Name attribute returns the new value.

Scenario:

Get attribute value.

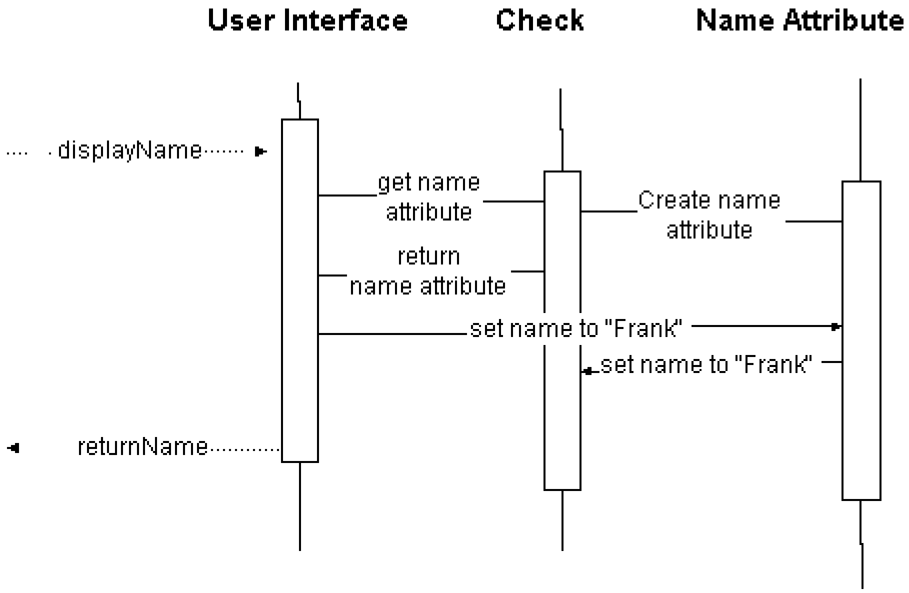


Stimuli:

1. User Interface receives a request to display attribute name.
2. User Interface asks Check for attribute name.
3. Check creates an editable attribute for Name.
4. Check returns name attribute to User Interface.
5. User Interface asks attribute for value.
6. Attribute sends getName to Check.
7. Check returns value of name to attribute.
8. Attribute returns value of name to User Interface.
9. User Interface returns value of name.

Scenario:

Update Attribute value.

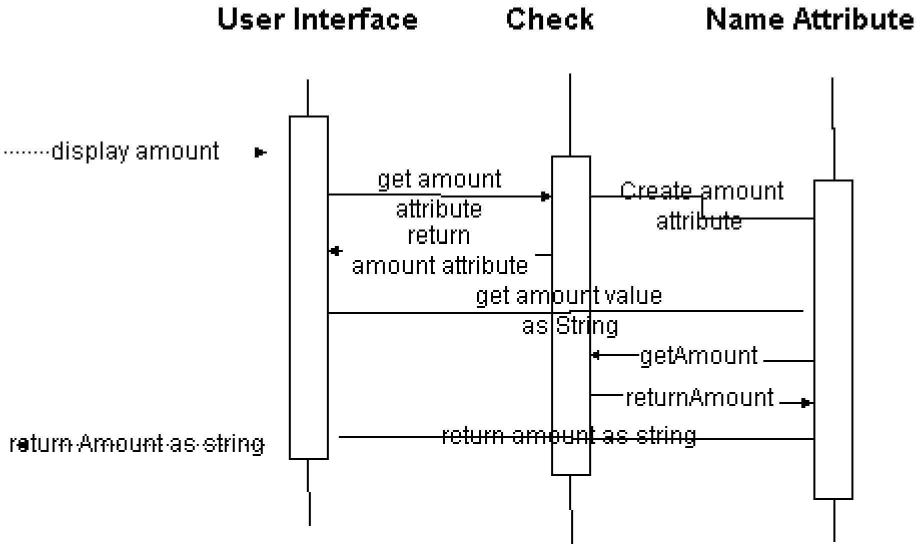


Stimuli:

1. User Interface receives a request to display attribute name.
2. User Interface asks Check for attribute name.
3. Check creates an editable attribute for Name.
4. Check returns name attribute to User Interface.
5. User Interface sends setValue to name attribute.
6. Attribute sends setName to Check.
7. Check sets the value of name to attribute to "Frank."

Scenario:

Display Attribute value as String.

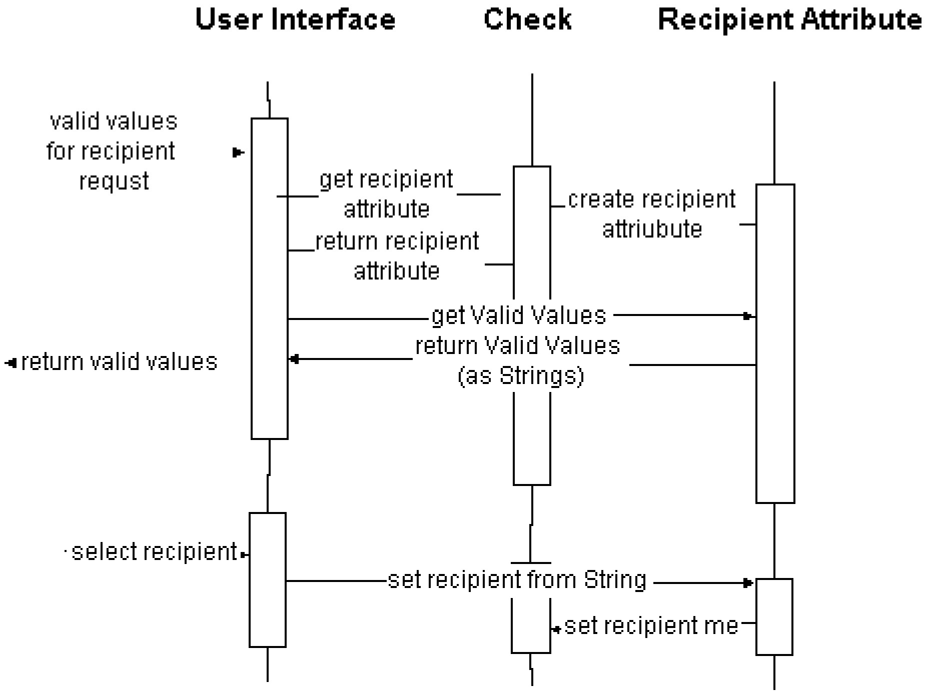


Stimuli:

1. User Interface receives a request to display attribute amount.
2. User Interface asks Check for attribute amount.
3. Check creates an editable attribute for amount.
4. Check returns amount attribute to User Interface.
5. User Interface asks attribute for value as a string.
6. Attribute sends getAmount to Check.
7. Check returns value of amount to attribute.
8. Attribute returns a string representing value of amount to User Interface.
9. User Interface returns string representing amount.

Scenario:

Get Valid values & set value from String.

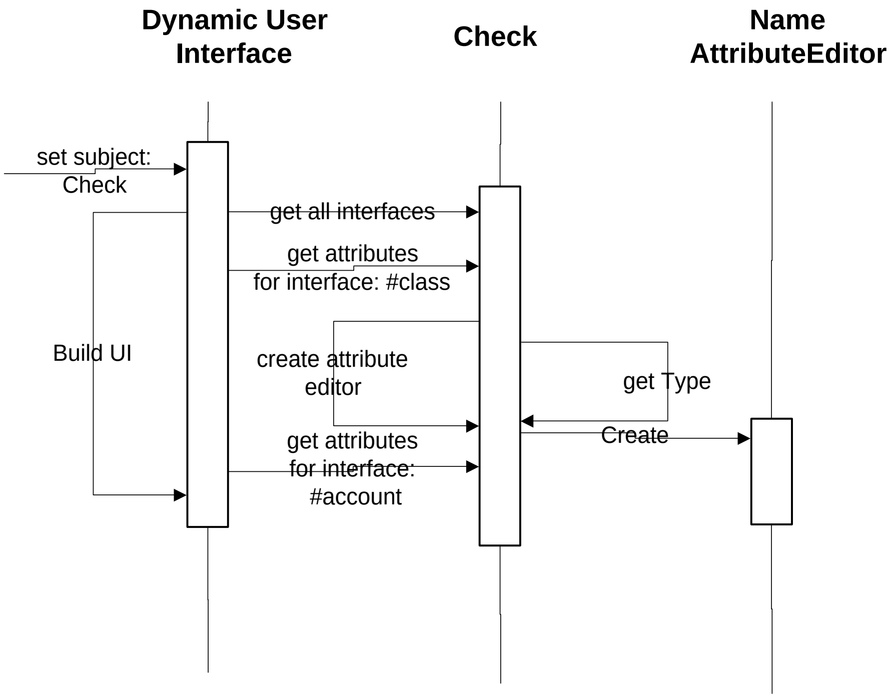

User Interface
Check
Recipient Attribute
valid values for recipient requst
get recipient attribute
create recipient attriubute
return recipient attribute
get Valid Values
return Valid Values (as Strings)
return valid values
select recipient
set recipient from String
set recipient me

Stimuli:

1. User Interface receives a request for valid values on attribute recipient.
2. User Interface asks Check for attribute recipient.
3. Check creates an editable attribute for recipient.
4. Check returns recipient attribute to User Interface.
5. User Interface asks attribute for valid values.
6. Recipient attribute returns a collection of strings representing the valid values to User Interface.
7. User Interface returns valid values.
8. User Interface receives a select recipient message. The selection is a string.
9. The User Interface sends setValueFromString to the recipient attribute.
10. The recipient attribute converts the string to the appropriate value and sends setRecipient with that value to the Check

Scenario:

Display all attributes of objects.



Stimuli:

1. A dynamic user interface is sent a set Subject request with the Check as the subject. This user interface will display all attributes of the Check as grouped into tabs according to their appropriate interactions.
2. The user interface asks Check to return all interactions.
3. Check returns a collection with two interaction names #Check and #account. All sub-components at minimum must support a class interface. The class interface defines those attributes which are explicitly defined on the sub-component's class.
4. The user interface then asks the Check for all the attribute editors on the Check interface.
5. The Check will check the type of each attribute on the Check interface and then create an appropriate editor for that attribute.
6. The Check returns the attribute editors to the user interface.
7. The Slider asks the user interface for all of the attribute editors on the #account interface.

Specification

Package: com.ac.eas.modeling

Related Frameworks

Framework: User Interface

Persistence Framework Overview

Purpose

The Persistence package provides a highly configurable layer of persistence services for business objects.

The Business Class Modeling / Persistence Framework comprises a number of interfaces, or roles, that must be performed in order to provide a framework for creating, retrieving, updating, and deleting business objects. The framework supports the following functions:

- Modeling a component's functional area of concern.
- Mapping business objects openly and flexibly to DBMS structures.
- Defining the application programming interfaces (APIs) that create, read, update, and delete business objects with integrity.
- Defining the basic persistence services (e.g., caching, object identity, lock management).

Many of the services can be enabled or disabled on individual business classes.
Services

Connections

The Connection service offers management of database connections. The service allows you to create, cache, and reuse connections to a Domain. Connections can be cached for a specified amount of time in order to minimize reconnection time.

Unit Of Work

The Unit Of Work allows the logical grouping of a series of related actions. For persistence, it can commit or rollback a sequence of persistence operations against a database connection.

Extent

The Extent facilitates manipulation of business objects which are mapped to database entities. Using an extent, a business object can be added, retrieved, altered, and / or removed from the database.

Record Locking

Record Locking allows you to specify whether optimistic locking will be supported by individual business objects. The operations provided in the Extent are invoked according to the locking policy of the object being operated on. The optimistically lockable objects are tracked until the Unit of Work is committed. In the event that data has been updated by another process between original record retrieval and Unit Of Work commit time, then a locking conflict handler is invoked on the associated business class. If pessimistic locking is supported by the business object, then the data store operations and locks are invoked directly.

Object Notification

Object Notification provides a set of methods that must be implemented by Persistable objects. They are invoked by the Extent and Unit Of Work. Business Objects can selectively implement logic in these methods to be invoked before or after various data store operations, e.g. add, update, remove, etc.

Object Identity

Object Identity guarantees usage of the exact same instance of a business class once it is retrieved from the database using the same Unit Of Work. Any subsequent retrievals of the same object, within the same Unit Of Work, will return the original instance instead of a new one. The original instance is retained and used throughout the life of the Unit of Work through which it was retrieved. Object Identity is automatically invoked along with the Extent retrieval service.

Business Object Caching

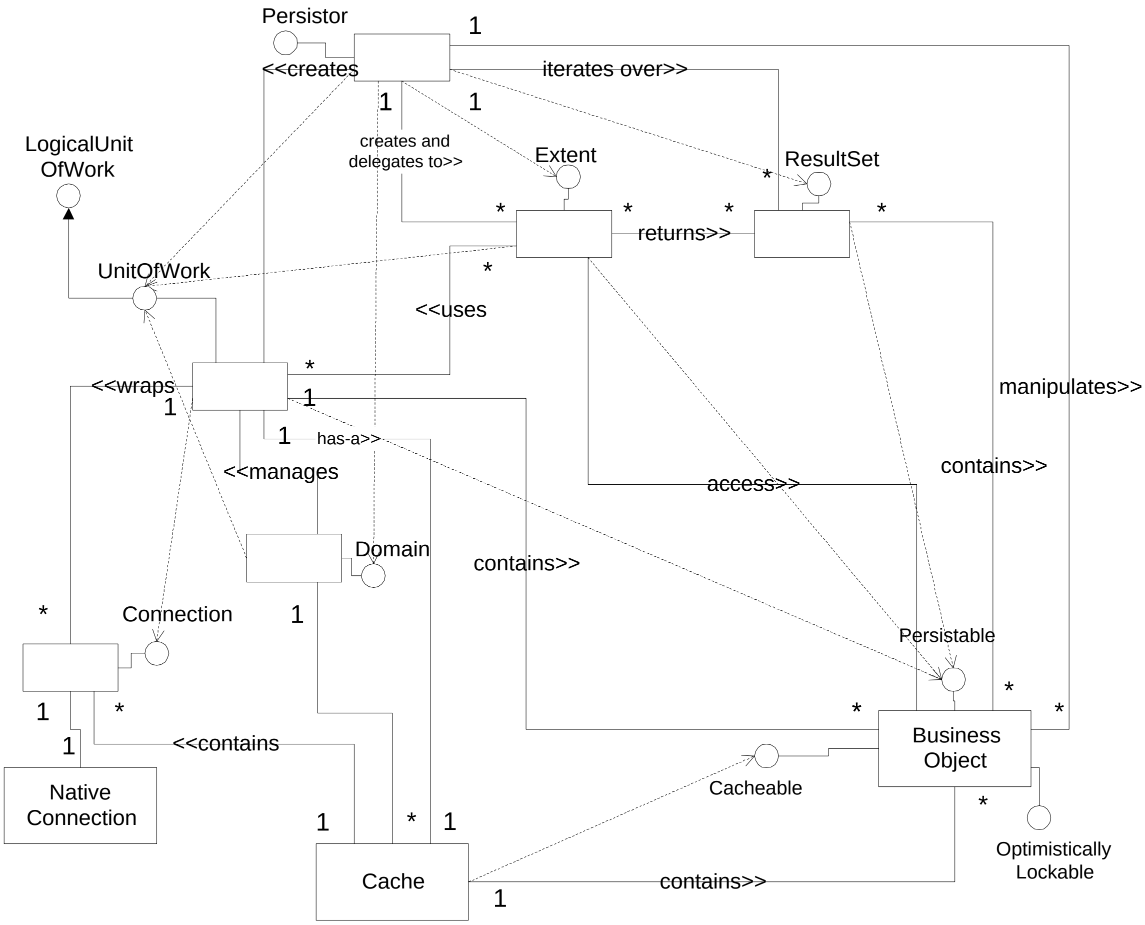
Business Object Caching allows you to support the ability to access previously retrieved business object instances in subsequent data store operations. If caching is supported by a business object, then when an an instance of the business object is initially retrieved, the instance is stored in a Cache in order to expedite subsequent retrievals .

Result Set

The Result Set allows you to easily iterate through the results of an object store query in an object-oriented manner.

Structure

Class Diagram



Participants

Persistor

- Implemented by objects whose functionality requires interaction with a data store.
- Creates extents for a specific Business Class.

Domain

- Implemented by classes which represent the physical data store, specified by an *object store* name, a *username*, and a *password.*
- Manages reuse of connections for the Connection service.
- Manages caching of business objects the Business Object Caching service.

Extent

- Implemented by objects that need to add, retrieve, alter, and remove Business Objects from an object store.
- Implementation objects can provide behaviors that maintain object identity policies.
- Implementation objects can provide behaviors that support optimistic locking for business objects which implement it. Relies on the operation's Unit Of Work to keep a trail of objects logically grouped into a unit of work.
- Implementation objects can provide behaviors that support object notification, calling back to the associated business object as appropriate when a data store operation is invoked.
- Implementation objects can utilize caching for accessing business objects which implement it.

Result Set

- Created from the results of an object store inquiry.
- Allows you to iterate through its contents.
- During iteration through the result set, notifies the business object to invoke required functionality.

Unit Of Work

- Implemented by classes which model a logical unit of work to be performed on a Connection to the data store. All work performed within the unit of work are either commited or rollbacked.
- Implementers must manage their own group of objects logically grouped into a unit of work to be processed together.
- Provides behaviors such as commit, rollback, checkpoint, and close.
- On checkpoint or commit, notifies the object of pending or completed data store operations to invoke required functionality.

Connection

- Implemented by classes which “wrap” a native connection to the data store
- Implementers encapsulate all the logic required to connect and disconnect from the data store.
- Provides behavior to perform operations on the native connection to the data store such as commit, rollback, and close.

Persistable

- Implemented by objects with corresponding data store entities.
- Provides a set of callback methods for the Object Notification service that can be implemented as required on an object.

OptimisticallyLockable

- Implemented by Persistable objects that can support optimistic locking.
- Provides a conflict handler which is invoked when data has been updated between retrieval and commit.

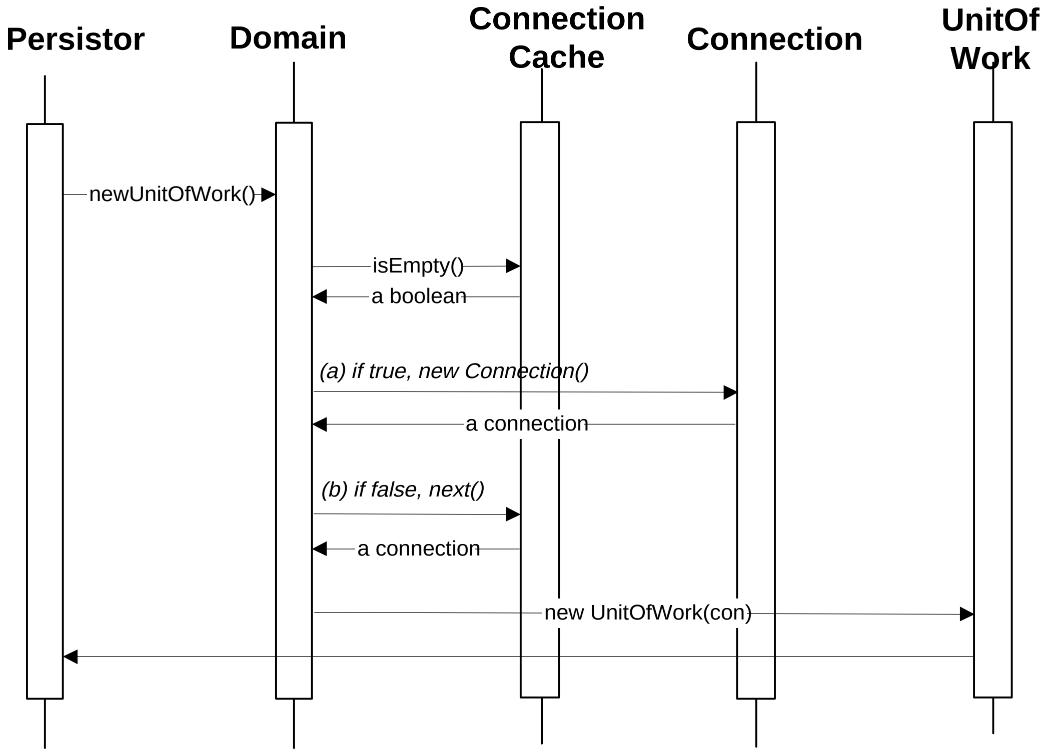
Cacheable

- Implemented by Persistable objects for which it is appropriate to cache instances from data store retrieval for future use.

Collaborations

Scenario:

Connection Service: creating and caching connections to the database.



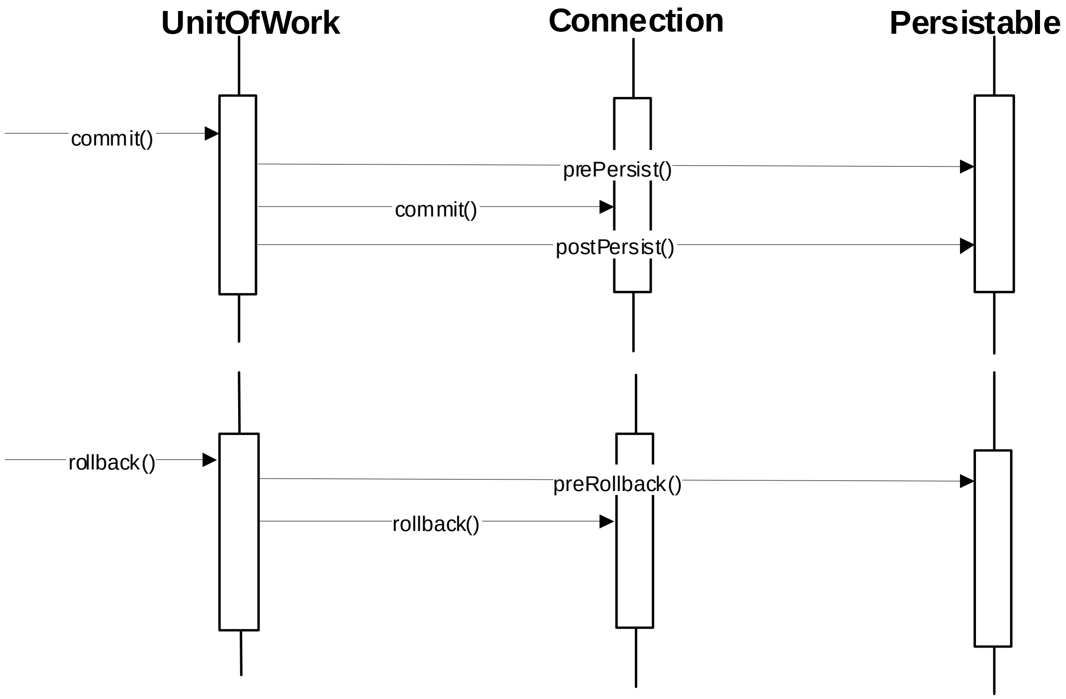
Stimuli:

1. A Persistor requests a new Unit Of Work from the Domain.
2. Domain checks to see if there are Connections cached in the connection cache. If true, the Domain requests the next cached Connection. If false, the Domain creates a new Connection, which creates a new physical connection to the database.
3. The Domain then creates a new Unit Of Work, passing the Connection. A new Unit Of Work instance is created each time, holding on to a Connection. The passed in Connection can be a cached Connection or a new Connection.
4. When the Unit Of Work is complete, the Connection that the Unit Of Work is holding on to is cached for reuse, and the Unit Of Work instance is destroyed. Only Connections are cached, not UnitsOfWork.

Scenario:

Unit of Work Service: committing / rolling back a unit of work.

Object Notification Service: notifying the business object that its corresponding data store entity has been added, updated, or removed.

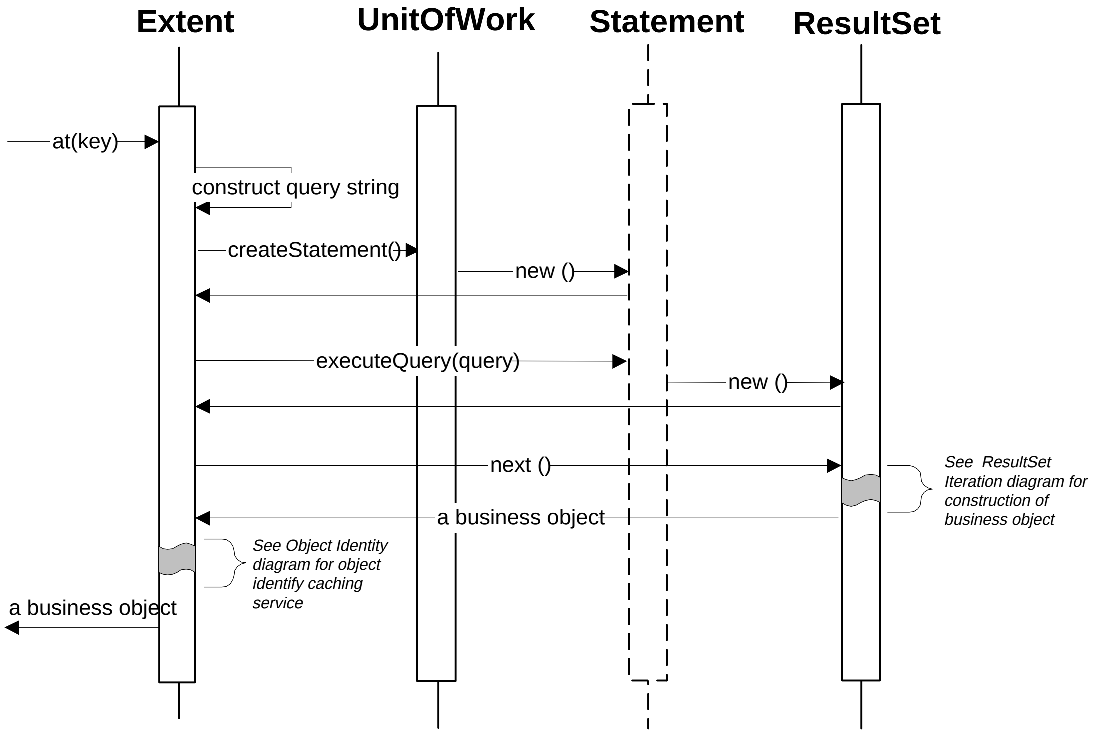


Stimuli:

1. At commit, the Unit Of Work sends a commit to the Connection its holding on to, which performs a commit on the native connection to the database.
2. The Unit Of Work also notifies the added / updated / removed objects before and after commiting the unit of work.
3. The Unit Of Work removes the Persistable object from the collection of objects it is tracking for Object Identity.

Scenario:

Extent Service : retrieving a business object from the database.

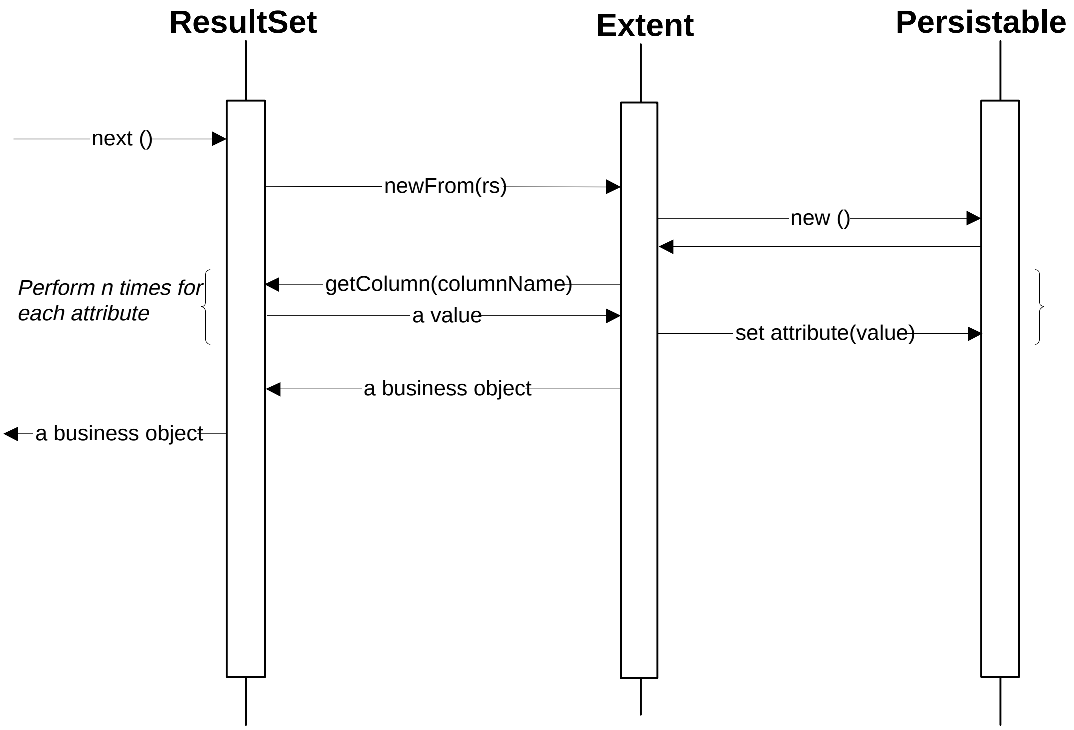


Stimuli:

1. A Persistor creates an Extent for a specific business class.
2. A retrieve operation (direct-key lookup of an object in this case) is issued via the Extent, for a Unit Of Work.
3. The Extent constructs the appropriate query and creates a statement to execute the query on the database.
4. A Result Set is constructed with the results from the query.
5. The result of the operation is returned to the Extent, which retrieves the first element of the Result Set and passes it back to the Persistor.

Scenario:

Result Set Service : iterating through the result set in an object-oriented manner.

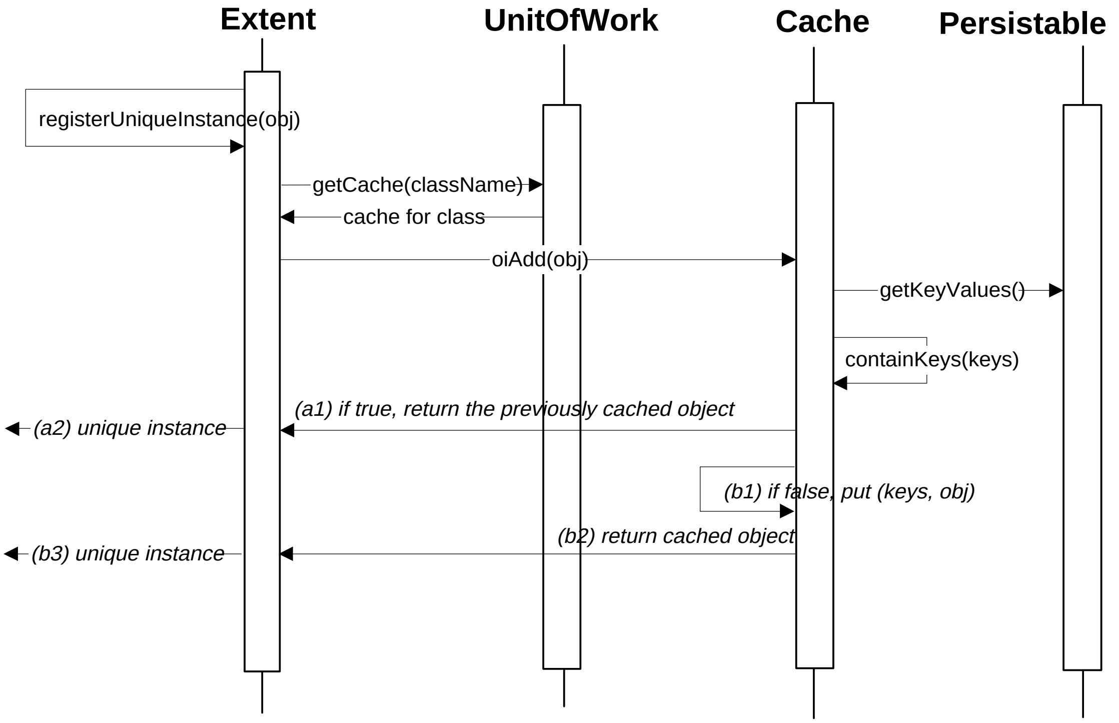


Stimuli:

1. After performing retrieve operation on an Extent, a Persistor gets back a Result Set.
2. Persistor sends next() to the Result Set to retrieve the next Persistable object in the Result Set.
3. Result Set then sends a message to the Extent object that performed the initial data store retrieval, passing in the current row of column values from the result set. The Extent object, since it is focused on a particular business class, then knows how to construct a new instance of the business class with the values of the current row.
4. The Extent passes the newly constructed business class instance back to the Result Set, which passes it to the Persistor. The Persistor gets back a new initialized business object.

Scenario:

Object Identity Service: guaranteeing usage of the exact same instance of a business class once it is retrieved from the database using the same Unit Of Work.

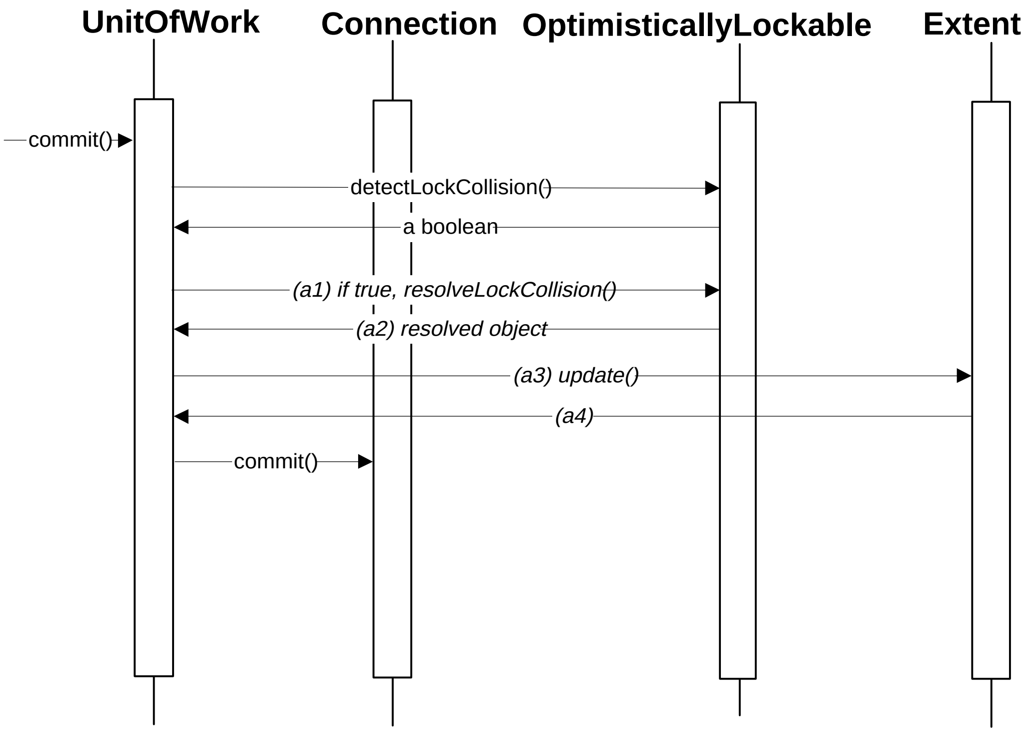


Stimuli:

1. After retrieving an instance of a business class, the Extent tries to register the instance as unique within the Unit Of Work that was used to retrieve the business object
2. The Extent looks up the Object Identity cache for the business class. A Unit Of Work holds on to a hashtable of business class caches keyed by class name.
3. The Extent then does an object identity add on the cache. The cache checks to see if an instance with the same key values as the object being added has already been added to the cache. If there is an object in the cache with the same keys, then that means that a unique instance of the business class has already been registered. That unique instance is then passed backed to the Extent, which now references that unique instance instead of the second instance. If the object being oiAdded is not in the cache, then the object is registered as a unique instance and then passed back to the Extent.
4. The Extent now references the only unique instance of the business class within the same Unit Of Work.

Scenario:

Record Locking Service : commit update for an optimistically lockable business object.



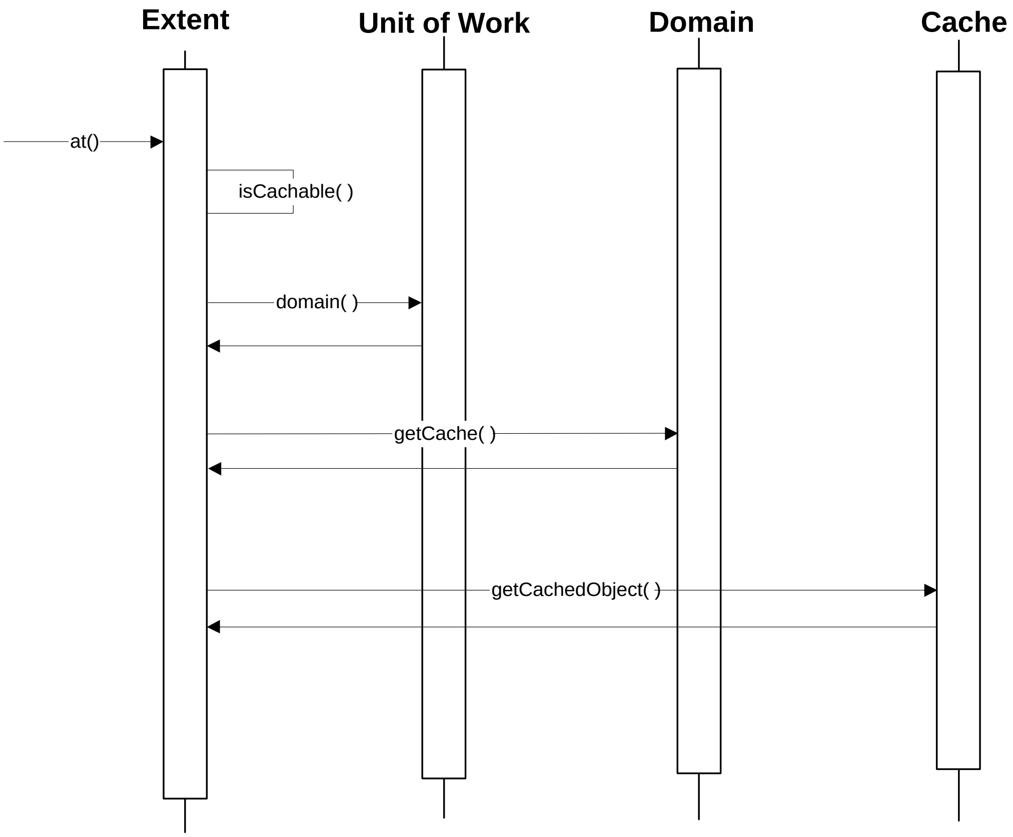
Stimuli:

In order to detect/resolve lock the object must be re-read from the extent.

1. At commit, the Unit Of Work checks for locking collisions for each optimistically lockable object in its trail.
2. The Unit of Work uses the connection to send the corresponding object store command through the Extent.

Scenario:

Business Object Caching Service : retrieving an object which has been previously retrieved on this connection.



Stimuli:

1. A Persistor requests that an Extent retrieve a business object.
2. The Extent checks to see if the business object is cachable.
3. If so, the Extent gets the Domain from the Connection.
4. The Extent gets the business object's Cache from the Domain.
5. The Extent requests the keyed object from the Cache.
6. If the business object was previously retrieved on this Unit Of Work, then the cached business object is returned from the Cache.

Specification

Package: com.ac.eas.persist

Related Frameworks

Framework: com.ac.eas.activity

Component Interface Framework Overview

Purpose

The Component Interface Framework is a framework for the implementation of custom components. The Component interface model separates the interface from the business logic. The Component encapsulates the business logic and a skeleton encapsulates the interface. A Proxy is used to communicate over the network to a skeleton.

Components are constructs that encapsulate and implement specific business functionality. Components tend to be larger than objects and are usually composed of many other constructs such as business objects, activity management frameworks, display frameworks, and integrated performance support. A component controls all access to its services and supports multiple interfaces or skeletons.

An interface (implemented by a skeleton) defines the operations, parameters, return types, and exceptions that can be raised. Skeleton objects are network addressable and can be defined in CORBA IDL or some other interface specification language. They are configurable with properties for use with a Trader Service. Lastly, skeletons unmarshal parameters associated with a request from another component and tie to a component implementation.

The proxy handles the marshalling of all the parameters required for a call as well as the ability to negotiate with a Naming or Trader Service.

Patterns

The Eagle Architecture Specification supports two similar yet distinct types of interfaces that differ in creation and scope.

Globally Addressable Interfaces

A Globally Addressable Interface (GAI) registers with a Naming or Trader Service and is globally accessible on the network. Any component can contact a Naming or Trader Service and gain access to a GAI.

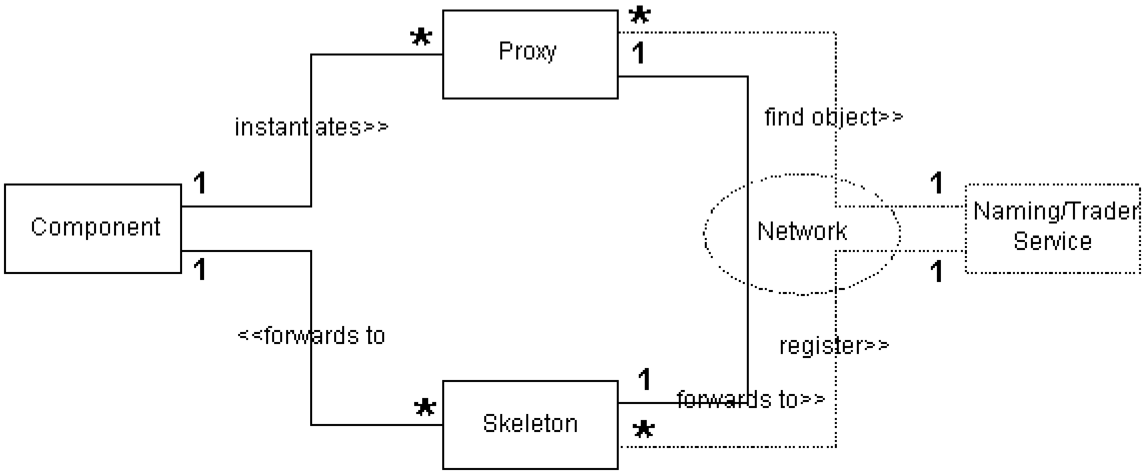
Locally Addressable Interfaces

A Locally Addressable interface differs from a Globally Addressable Interface in respect to its addressability. A Locally Addressable interface does not register with the Naming Service and therefore has no public network addressability. A Locally Addressable interface is only obtainable through a Globally Addressable Interface that returns an object reference.

A Globally Addressable interface registers with a Trader or Naming Service and has network addressability (see scenario below).

Structure

Class Diagram



Participants

Proxy

- Uses a Naming or Trader service to locate and connect to a skeleton somewhere on the network
- Forwards messages to a corresponding skeleton elsewhere on the network
- Forwards replies back to the client component

Skeleton

- Registers with a Naming or Trader Service
- Specific to the services a component chooses to provide or serve
- Addressable on the network
- Forwards messages to the component for processing

Component

- Encapsulates specific business logic
- Different components encapsulate different information
- Control information flow to and from its business objects

Naming/Trader Service

- Can be a Naming or Trader Service <provide link to services>
- Holds object references to instances of the skeleton class
- Provides a directory service to "look up" skeleton objects

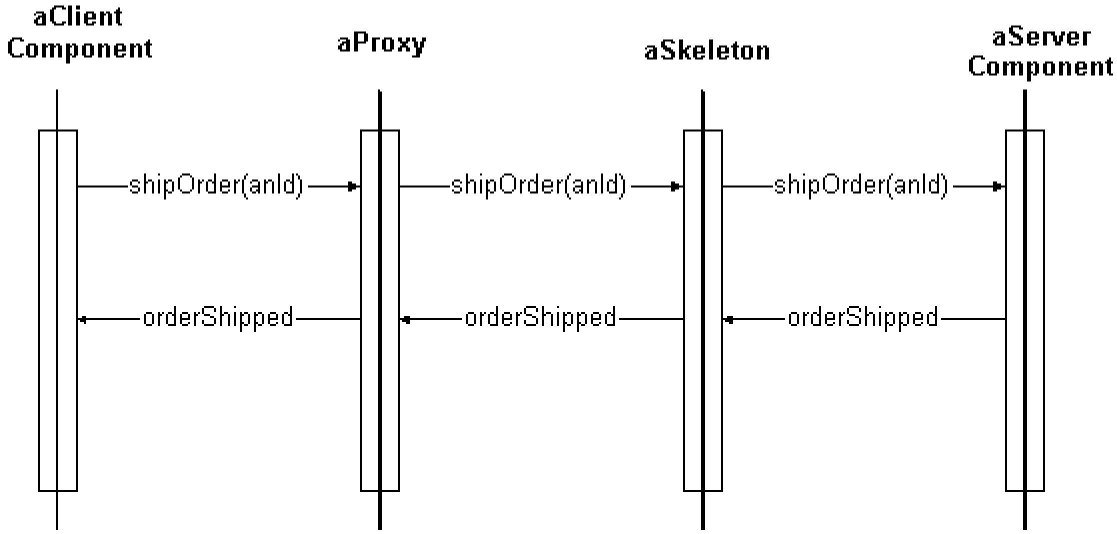
Network

- Some sort of LAN or WAN network

Collaborations

Scenario:

A client component requests that a server component ship an order. (This scenario assumes that the Proxy has already "connected" to a Skeleton through a Trader or Naming Service.)



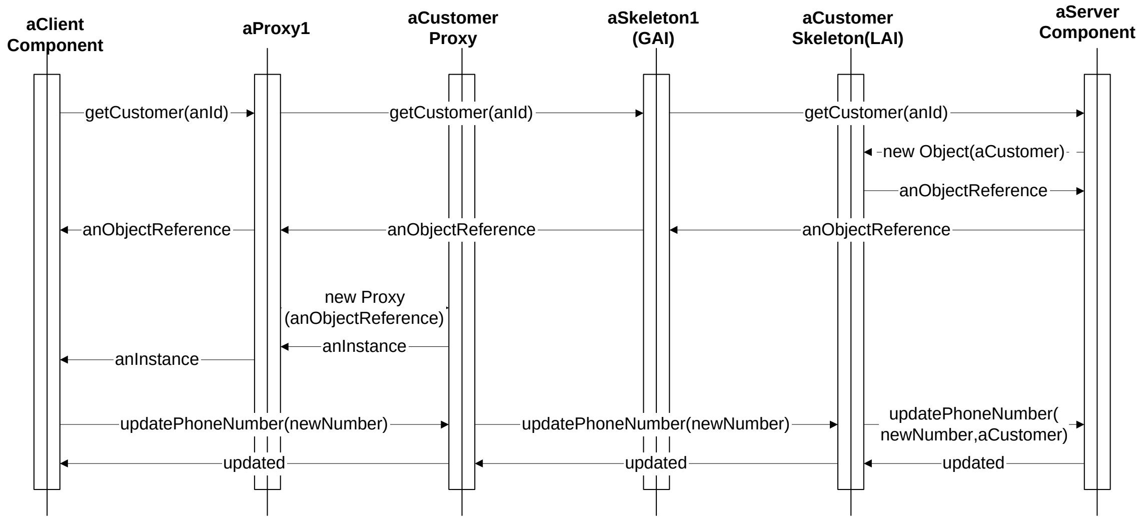
Stimuli:

1. aClientComponent sends a shipOrder with anId to aProxy.
2. aProxy forwards shipOrder with anId to aSkeleton.
3. aSkeleton forwards shipOrder with anId to aServerComponent.
4. aServerComponent looks up the order associated with anId and successfully ships the order.
5. aServerComponent returns an "orderShipped" return value through aSkeleton and aProxy to aClientComponent.

Scenario:

A client component updates the phone number on a customer profile. This scenario demonstrates the creation and use of a Locally Addressable Interface.

This scenario also assumes that the Proxy (aProxy1) has already "connected" to a Skeleton through a Trader or Naming Service.



Stimuli:

1. aClientComponent sends a getCustomer with anId to aProxy1 (Note: aProxy1 already connected to a skeleton using a Naming or Trader Service).
2. aProxy1 forwards getCustomer with anId to aSkeleton1.
3. aSkeleton1 forwards getCustomer with anId to aServerComponent.
4. aServerComponent gets the customer information associated with anId from the database. aServerComponent instantiates a customer based upon the database information.
5. aServerComponent instantiates aCustomerSkeleton. aCustomerSkeleton is a Locally Addressable Interface to an instance of a customer (the one from the database).
6. aServerComponent returns anObjectReference the Locally Addressable Interface aCustomerSkeleton through aSkeleton1 to aProxy1.
7. aProxy1 instantiates aCustomerProxy based upon anObjectReference.
8. aProxy1 returns an instance of aCustomerProxy to aClientComponent.
9. aClientComponent sends updatePhoneNumber with newNumber to aCustomerProxy.

10. aCustomerProxy forwards updatePhoneNumber with newNumber to aCustomerSkeleton
11. aCustomerSkeleton passes updatePhoneNumber with newNumber AND aCustomer to aServerComponent.
12. aServerComponent passes updatePhoneNumber with newNumber TO aCustomer (not shown in picture).
13. aCustomer updates the phone number to newNumber.
14. aCustomer returns an "updated" return message through aServerComponent, aCustomerSkeleton and aCustomerProxy to aClientComponent.

Specification

Package: com.ac.eas.java.compinter

Related Frameworks

Framework: Naming Service

Framework: Trader Service

Login Framework Overview

The Login framework defines the operations and the policies relative to a user logging in to an application. The objective of the Login framework is to provide mechanisms to check the true identity of users and to give them a way to be identified within an Enterprise System. The Login framework encapsulates the user authentication procedures, thereby isolating the applications from their details. Moreover, the Login framework provides the means to retrieve a set of information that describes the user, including security information.

An application, at startup or on-the-fly, which needs a user (and his/her Context) calls the Login framework for '*login.*' Before closing, the application must perform a '*logoff.*'

The requirements for, and the implementation of the "one-time-login" or the "on-the-fly-login" might vary. The Login framework has been designed to support this variety and flexibility.

There are different ways of authenticating a user logging on to an application:

- User identifier and passwords
- Smartcard and PINs (Personal Identification Numbers)
- Biometrics; for example, fingerprints or signatures

Passwords and user identifiers can be intercepted if they are transmitted in a clear format; a more secure method is to encrypt them. Encrypting sensitive information is a primary technique used to protect it from intruders.

The use of smartcards combines "something you have" "and something you know" types of information to obtain enhanced security.

In the case of a smartcard, two kinds of authentication should be carried out:

- *User-to-Card Authentication*, which uses a PIN code (or biometric). The user enters his/her PIN or biometric, which is then checked against the one stored on the card. This local transaction between the card and the user is a very secure.
- *Card-to-Server Authentication*. The server generates a random number, also called a *challenge,* and sends it to the card. The card encrypts the challenge using its private key and sends it back to the server. The server decrypts the challenge using the client's public keys and checks the result against the original challenge. If they match, the card has been authenticated.

Services

Within this document:

- (the object) *LoginInfo* is used to define information that authenticates a user; for example, User ID + password or PIN + Smartcard-check
- (the object) *User* or *Context* represents information relative to a user
- *Main User* is used to identify the user who first logged on to a Client machine

Context

The Context object maintains all the information relative to the environment of a logged user. One Context is created for every new logged user. The Context allows to retrieve:

- *User information*: for example the User ID, his / her roles, his / her preferences (language, colors, etc.) can be accessed in the User reference handled by a Context
- *Environment information*: more dynamic information can be maintained in a Context, for example temporary changes of user's preferences, current location of a logged user (Web, office LAN, enterprise WAN, etc.), applications being accessed by a user, etc.

In a distributed system, resources can be spread over several machines and shared among several users. Identifying who actually accesses a resource becomes a specialized operation. The Context is responsible for providing the current user identification to all applications collaborating across a distributed system.

Get Roles

This service retrieves the role(s) of a logged user. The collection of Roles for a logged user is critical information for controlling user access to secured and / or audited resources (refer to Framework Security Policy Management).

Application User Manager

Objects implementing this interface are the *entry point* of the entire Login framework. Each application must create and maintain its own dedicated Application User Manager.

This relationship can be implemented in different ways:

- Every application is defined in its own memory space (a Virtual Machine); in this case, the Application User Manager can be implemented as a singleton.
  For further information about the singleton pattern see: *"Design Patterns" Gamma, Helm*
- Multiple applications are defined in the same memory space; in this case, the application maintains a physical reference to its own unique instance of an Application User Manager object.

The Application User Manager holds a reference to its current Context object. This object provides the following services, which allow applications to authenticate and log in users and to obtain the Context associated with the user:

Login

This service attaches the invoking application to a user (and his/her Context). This service is typically invoked when an application starts. The Application User Manager checks that it is not already maintaining a Context; if this is so, an exception is raised. The method Login is then invoked on the Client User Manager. This type of login does not return any parameters, but it can raise an exception.

Login-on-the-fly

This service attaches the invoking application to a user (and his/her Context). This service is typically invoked when an application needs to invoke a user with different access rights for a short time.

The Application User Manager checks that it is already maintaining a Context; if it is not, an exception is raised. Note that several login-on-the-fly logins can be run at the same time. This type of login does not return any parameters, but can raise an exception.

Logoff

This service detaches the invoking application from the user (and his/her Context) to which it is currently attached. This service is typically invoked when an application closes. The Application User Manager checks that it is already maintaining a Context; if it is not, an exception is raised. If one or many login-on-the-fly logins have been created for the current application, the logoff will automatically close all of them before closing the main user login session. The logoff does not return any parameters, but can raise an exception.

Logoff-on-the-fly

This service detaches the invoking application from the on-the-fly user (and his/her Context) to which it is currently attached. It then attaches the session to the user who was active before the last ‘login-on-the-fly’. This service is typically invoked when an application closes a login-on-the-fly session. The Application User Manager checks that the current user is actually an on-the-fly user. If the user is not, an exception is raised.

This service does not return any parameters, but can raise an exception.

Get Context

This service returns the *Context* object of the user currently attached to the Application User Manager, and therefore to its application.

Client User Manager

Objects implementing this interface centralize and manage the collection of all the users logged on to a Client machine. Since there is only one Client User Manager per client machine (as opposed to one Application User Manager per application) these users can be shared across applications.

This is required to support the one-time-login, needed so that a user does not have to identify themselves every time they start a new application.

The Client User Manager object collaborates with an object implementing the User Manager Policy interface in order to support both general and policy-specific behavior. The Client User Manager is responsible for the general behavior.

Depending on the implementation choices of the applications and their Application User Managers, the Client User Manager can be implemented:

- In the same memory space as the Application User Managers
- In its own memory space, where it can be accessed transparently by the Application User Managers via object-based middleware (for example, a CORBA-compliant ORB)

This object provides the following services in its interface:

GetAPrimaryLoggedUser

This service returns the Context of a user (see above). The Client User Manager invokes the User Manager Policy to determine if a user has already logged-in, and if the Login can be reused. If so, the Client User Manager returns the Context returned by the User Manager Policy. If not, the Client User Manager invokes the Login Controller to login a new user. The returned Context is then passed to the User Manager Policy to be retained and returned to the invoking Application User Manager.

The GetAPrimaryLoggedUser service can also raise an exception.

GetASecondaryLoggedUser

This service returns the Context of a user (see above). The Application User Manager calls this method when it receives a login-on-the-fly request. The Client User Manager then invokes the User Manager Policy to determine if a user has already logged-in, and if the Login can be reused. If so, the Client User Manager returns the Context returned by the User Manager Policy. Otherwise, the Client User Manager invokes the Login Controller to login a new user; the returned Context is then passed to the User Manager Policy to be memorized and returned to the invoking Application User Manager.

The GetASecondaryLoggedUser service can also raise an Exception.

RemoveLoginFromAUser

This service manages the fact that the passed user (i.e., his / her Context) is about to be used by one less application. Since the list of logged users is managed by the User Manager Policy, the message is directly passed to it.

This service does not return any parameters, but can raise an exception.

User Manager Policy

An object implementing this interface collaborates with the Client User Manager to support both general and policy-specific behavior. The User Manager Policy is responsible for the policy-specific behavior.

The following examples of policy-specific behaviors assume that:

- The user U1 has launched the applications A1 and A2.
- The user U2 has logged-on-the-fly onto A2.

Examples of policy-specific behavior are:

- The application A3 is launched, and a Login is required:
- The User Manager Policy automatically returns the main user.
- The User Manager Policy presents the list of users that are already logged in. The user selected from this list becomes the current user attached to A3.
- A Login-on-the-fly is required; note that it does not matter which application requires it:
- The User Manager Policy presents a list of users that are already logged in. If a user is selected, that user is returned as the Login-on-the-fly user; otherwise, the Policy returns no Context and the Client User Manager invokes the Login Controller. The User Manager Policy never returns a Context, so the Login Controller is always invoked.

The User Manager Policy associates a counter to each logged user that it manages. This counter is incremented every time an Application User Manager is attached to a user, and decremented every time it is detached. When this counter is equal to zero, it means that no applications are logged to the associated user, and the user can be removed from the list managed by the User Manager Policy.

The User Manager Policy provides the following services in its interface:

GetAPrimaryLoggedUser

This service returns the *Context* object representing one of the users managed by the User Manager Policy. Depending on its policy-specific behavior, it might present a list of users so that one can be selected. If no user is currently managed or selected from the presented list, an exception is raised.

GetASecondaryLoggedUser

This service returns the *Context* object representing one of the users managed by the User Manager Policy. Depending on its policy-specific behavior (in particular on the policy for users that are logging on-the-fly) it might have to present a list of users so that one can be selected. If no user is currently managed or selected from the presented list, an exception is raised.

AddLoginToUser

This service increments the counter associated with the passed user (his / her Context). If the passed user cannot be added, an exception is raised.

RemoveLoginFromUser

This service decrements the counter associated with the passed user (his / her Context). If this counter then equals zero, the User Manager Policy might decide to remove the user from the collection of Logged Users it manages. If the passed user cannot be found in the collection, an exception is raised.

Login Controller

An object implementing this interface encapsulates all the behaviors required to authenticate a new user.

Login

This service allows a user to enter the required Login information and validate it. Note that the Login Validator records the login information to check that the same user does not login several times. If successful, a User Identifier is returned; otherwise an exception is raised.

Logoff

This service notifies the Login Validator that the user is about to logoff.

Context Builder

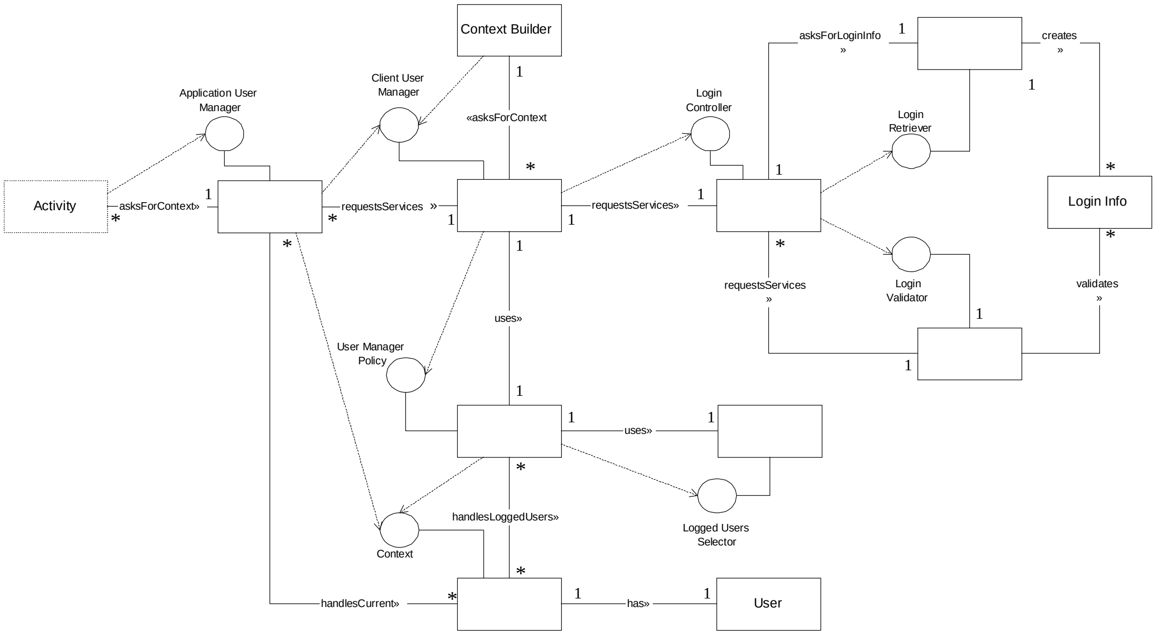
The Context Builder interface is responsible for building a new Context object.

BuildContext

This service builds a new Context object based on the passed User Identifier. The Login Controller typically returns the User Identifier (see above).If successful, a Context object is returned; this object includes a reference to a User Profile Object.

Structure

Class Diagram



Participants

Application User Manager

- This object is the *entry point* for the entire Login framework.
- Each application must create and maintain its own dedicated Application User Manager.

Client User Manager

- This object centralizes and manages the collection of all users logged on to a Client machine. As there is only one Client User Manager per client machine (as opposed to one Application User Manager per application) these users can be shared across applications.

User Manager Policy

- This object collaborates with the Client User Manager to support both general and policy-specific behavior. The User Manager Policy is responsible for the policy-specific behavior.

Login Controller

- This object encapsulates all the behaviors required for the authentication of a new user.

Login Retriever

- This object is responsible for collecting the Login information specified by the user. Typically this could be a UI where a user could enter either his/her User ID and password, or a PIN associated with the user’s smartcard. This object is typically implemented on a client machine.

Login Validator

- This object is responsible for validating the Login information entered by the user (see Login Retriever) and uses information maintained centrally on the server. This object is typically implemented on the server machine.

LoginInfo

- This object maintains the Login information specified by a user to identify himself/herself.

Context

- This object is responsible for maintaining the information relative to a logged user.

User

- This object was originally defined in the IPS (Integrated Performance Support) framework, and has been slightly modified in the current framework.

Role

- Refer to Framework Security Policy Management

Context Builder

- This object is responsible for building a new Context object.

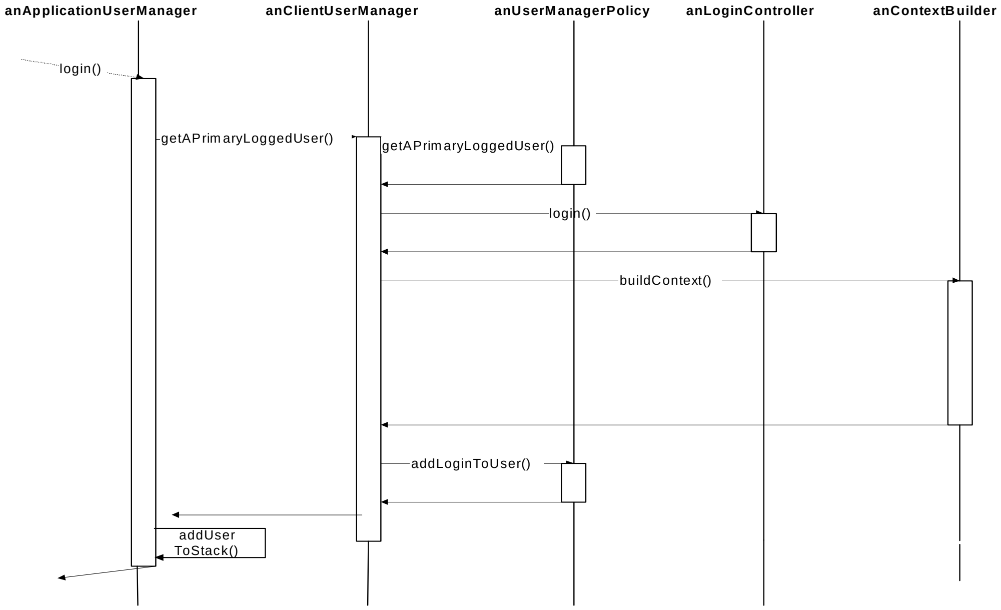
Logged User Selector

- This object is responsible for presenting a list of logged users; the person working on the Client machine can then select one.

Collaborations

Scenario 1:

An application, at startup, invokes the login method of the Application User Manager which, in turn, calls the login method of the Client User Manager. The scenario shows the case in which the user has to be authenticated.



Stimuli:

The Application calls the login method of the Application User Manager, which then calls the login method of the Client User Manager.

1. The Client User Manager calls the getAPrimaryLoggedUser method of the User Manager Policy. In this case, it raises an exception to indicate that no user (login) can be reused.

2. The Client User Manager calls the Login Controller object to have the user authenticated.

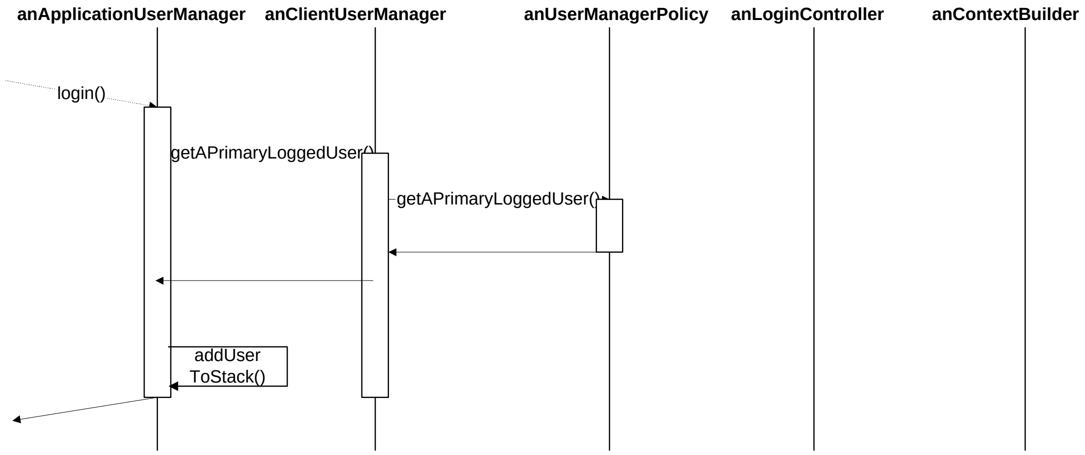
3. The Client User Manager calls the Context Builder to have it create a *Context* object. The *Context* object then creates a *User* and initializes its objects.

4. The Client User Manager passes the Context to the User Manager Policy, which then updates its list.

5. The Client User Manager returns the Context object to the Application User Manager, which will make it available to the application.

Scenario 2:

This scenario is similar to Scenario 1 (see above). The difference is that here the User Manager Policy returns an already-logged user; typically logged and used in a different application.



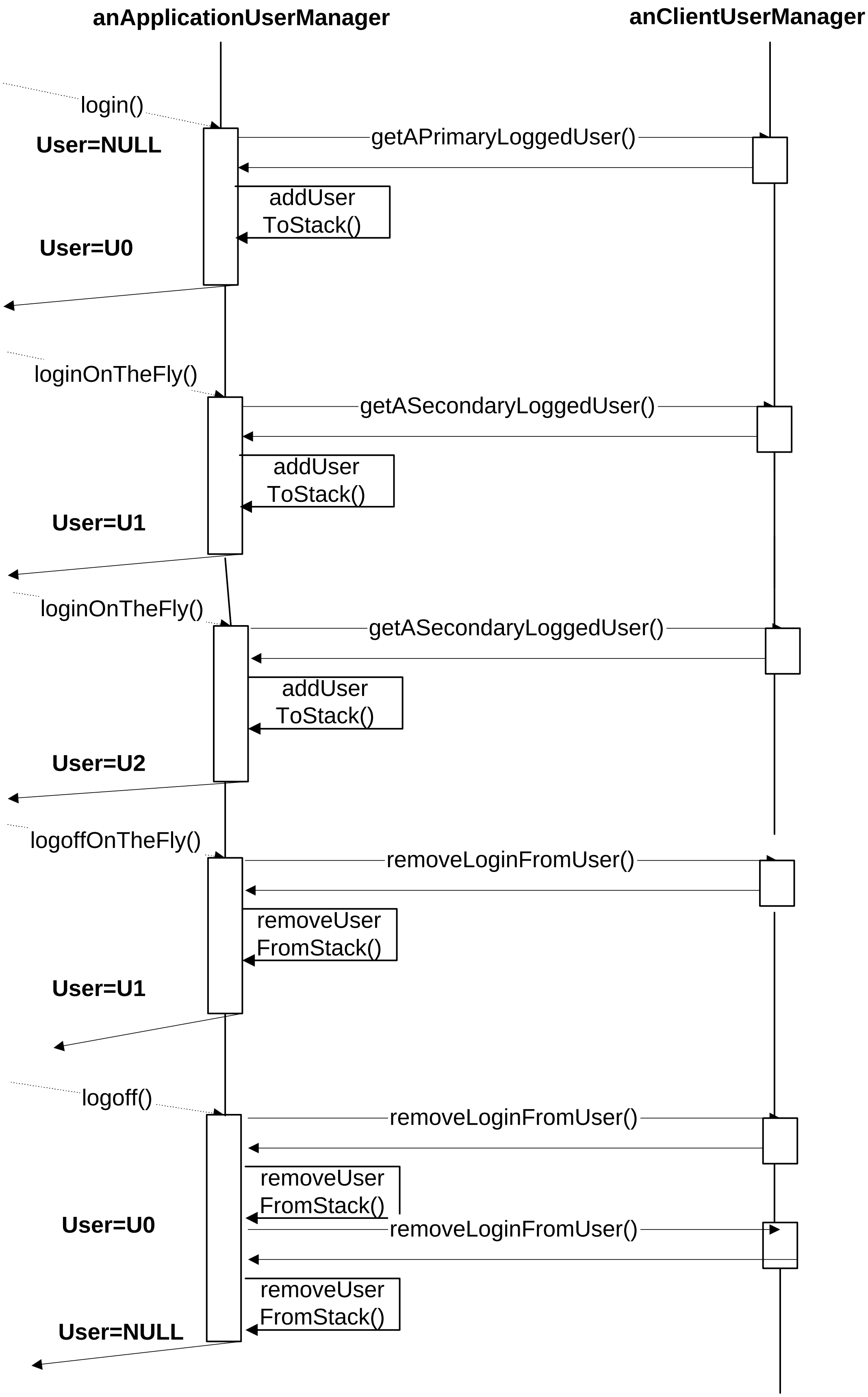
Stimuli:

The Application calls the login method of the Application User Manager which, in its turn, calls the login method of the Client User Manager.

1. The Client User Manager calls the getAPrimaryLoggedUser method of the User Manager Policy. In this case, it returns a full Context object.
2. The Client User Manager returns the Context object to the Application User Manager that will make it available to the application.

Scenario 3:

This scenario depicts how the Application User Manager manages its stack of Logged Users, and how it collaborates with the Client User Manager to keep the central Logged User list up-to-date.


anApplicationUserManager
anClientUserManager
login()
User=NULL
getAPrimaryLoggedUser()
addUser
ToStack()
User=U0
loginOnTheFly()
getASecondaryLoggedUser()
addUser
ToStack()
User=U1
loginOnTheFly()
getASecondaryLoggedUser()
addUser
ToStack()
User=U2
logoffOnTheFly()
removeLoginFromUser()
removeUser
FromStack()
User=U1
logoff()
removeLoginFromUser()
removeUser
FromStack()
User=U0
removeLoginFromUser()
removeUser
FromStack()
User=NULL

Specification

Package: com.ac.eas.security.login

Related Frameworks

Framework: Secured UI

Role Delegation Framework Overview

Purpose

The Role Delegation framework provides services to transfer an Access Right from a source User to a target User. This transfer must always be for a specific and limited purpose. The Role Delegation enables someone to act on behalf of someone else.

Some restrictions in the Role Delegation can be used:

- A time period can be applied to the delegation of Role.
- The transfer of a role can also be restricted to a limited number of resources.

These restrictions are expressed by constraints that need to be applied to the delegated role. These constraints limit the transfer of the role to a time period and a number of Access Rights.

Three delegation schemes are supported by the Role Delegation Framework. A User can transfer his/her Access Rights, during some validity period to:

- One property or method on one object by transferring a single Rule.
- All properties and methods on one object by transferring every Rules relative to the Securable Class.
- All Access Rights of a given Role by transferring all Rules of this Role.

Design Goals

The Role Delegation Framework provides simple and configurable mechanisms to:

- Let a source User transfer his/her Access Rights to a target User.
- Define a set of constraints that will restrict the time period and the number of resources to be transferred.

The Role Delegation Framework does not cover the mechanisms to execute the role transfer. For example:

- Single application started on the client machine.
- Administrative application used to manage the security policy of the Enterprise Information System.

Services

Role Transfer: Delegation Instantiater

The Delegation Instantiater Object is invoked when a source User wants to delegate one of his / her Role to a target User.

It defines the following mechanisms:

- Identifying the Role that the source User wants to transfer.
- Identifying the constraints that need to be applied to the transferred Role. These constraints are the following:
  - *Transfer Validity Period*. The transfer of the Role will only be valid during a period of time defined by a start date and an end date.
  - *Applicable Class*. The transfer of the Role will only be valid for one particular Class.
  - *Applicable method of an Applicable Class*. The transfer of the Role will only be valid for one property or one method of a particular Class.
- Creating the Delegated Role.
- Creating all the Delegated Rules that correspond to the identified transfer constraints.
- Populating the newly created Delegated Role with these Delegated Rules.
- Adding the Delegated Role to the target User.

The Delegation Instantiater must be invoked by any applications that execute a Role Delegation operation. It can be for example a system administration application that manages the User of the Enterprise Information System.

Transfer Period Validation: Delegation Housekeeper

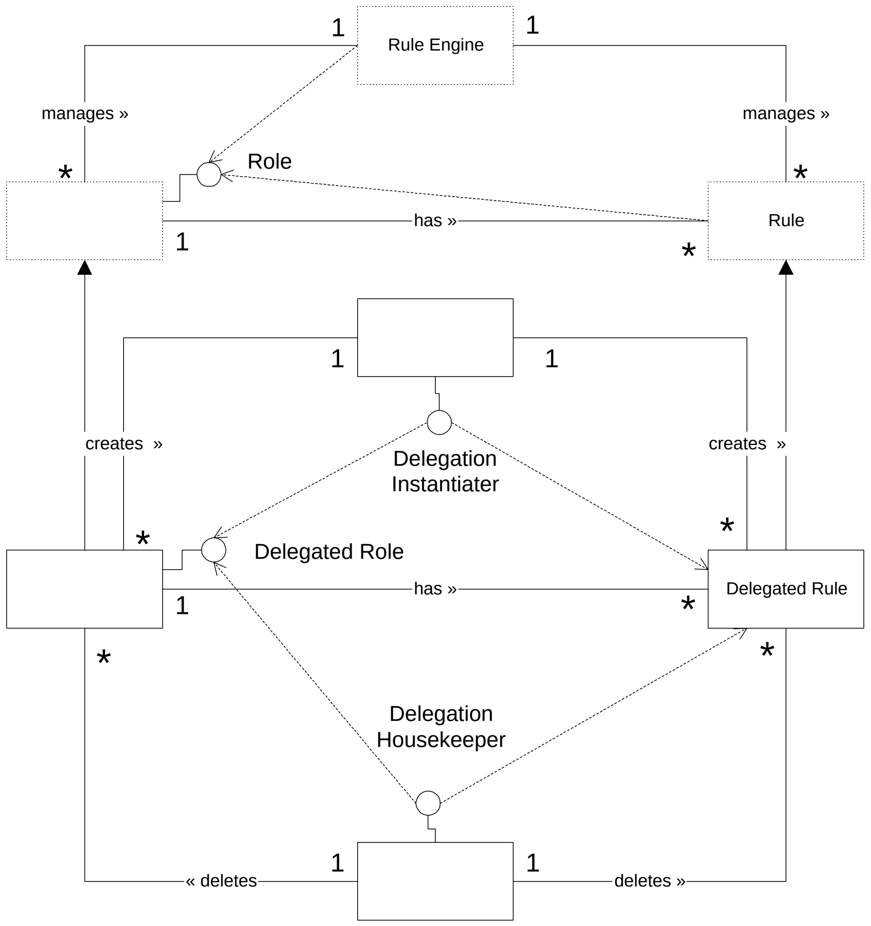
The Delegation Housekeeper is responsible for:

- Checking if the validity period of a Delegated Rule has expired.
- Removing the expired Delegated Rule from the Delegated Role.
- Removing the Delegated Role if all its Delegated Rules of the Delegated Role have been deleted.

The Delegation Housekeeper can be invoked by a batch application running at a pre-defined frequency to clean-up the expired Delegated Rule.

Structure

Class Diagram



Participants

Role

- Identifies a role that a user can play in the system (see the *Security Policy Framework*).

Rule

- Maintains the Access Right information (see the *Security Policy Framework)*.

Rule Engine

- Maintains a list of applicable rules (see the *Security Policy Framework*.

Delegation Instantiater

- Identifies the Constraints that need to be applied to the Role Transfer. The Constraints can be of three types:
  - *Period Validity Constraint*. The transfer of a Role is only valid during a certain period of time.
  - *Class Applicability Constraint*. The transfer of a Role is only applicable for a certain Class.
  - *Member's Class Applicability Constraint*. The transfer of a Role is only applicable for a certain property or method of a Class.
- Constructs the Delegated Role.
- Constructs all the Delegated Rules that correspond to the identified transfer Constraints.
- Populates the newly created Delegated Role with these Delegated Rules.
- Adds the Delegated Role to the target User.

Delegated Rule

- Contains a Period Validity Constraint (see the *Security Policy Framework*.

Delegated Role

- The Delegated Role is a Role (see the Security *Policy Framework*) that contains all the Delegated Rules.

Delegation Housekeeper

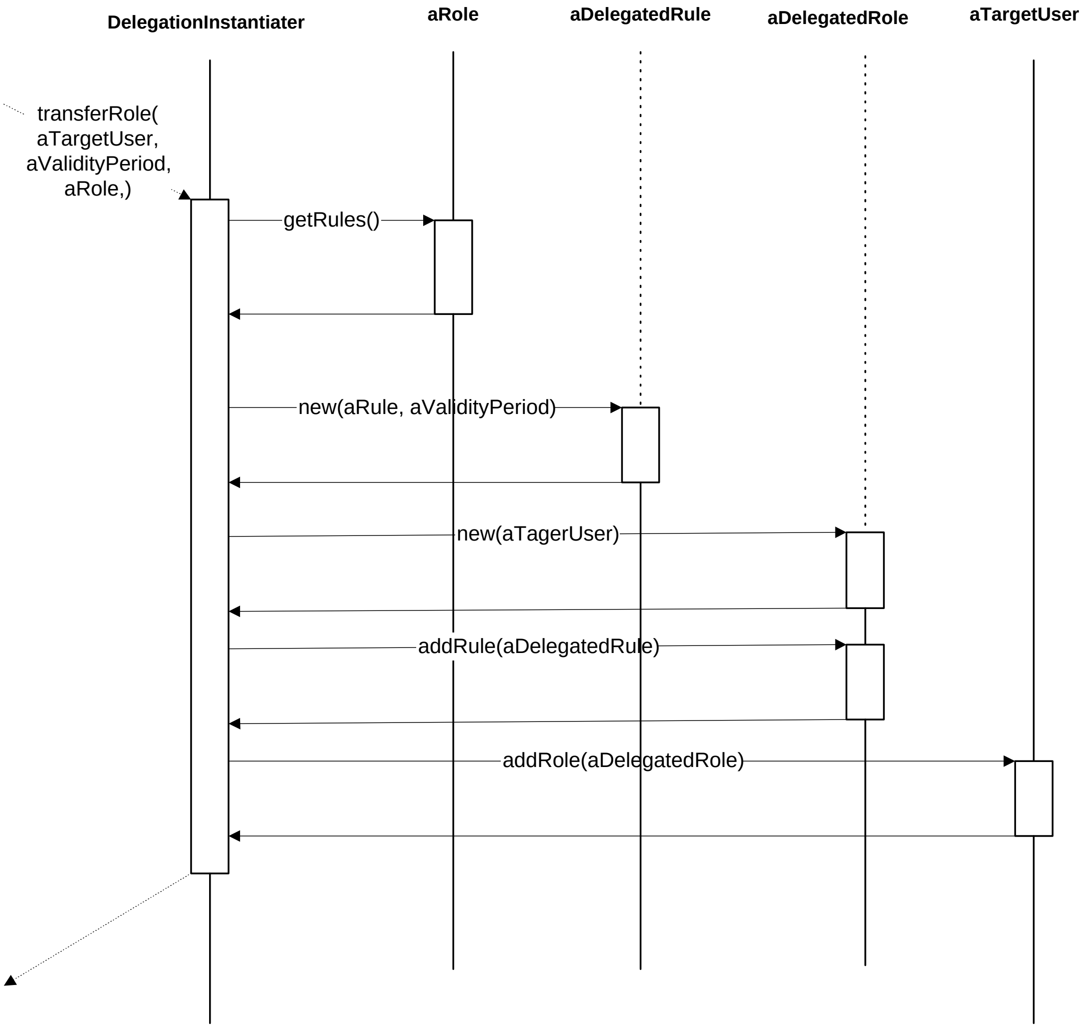
- Checks if the validity period of a Delegated Rule has expired.
- Removes the expired Delegated Rule from the Delegated Role.
- Removes the Delegated Role if all its Delegated Rules of the Delegated Role have been deleted.

Collaborations

Scenario 1:

Transfer all Rules of a Role from a source User to a target User.

The following diagram shows the flow of requests between the objects involved when the transferRole of the DelegationInstantiater is called. The transferRole method is used to transfer all the Access Rights associated to a Role from a source User to a target User.



Stimuli:

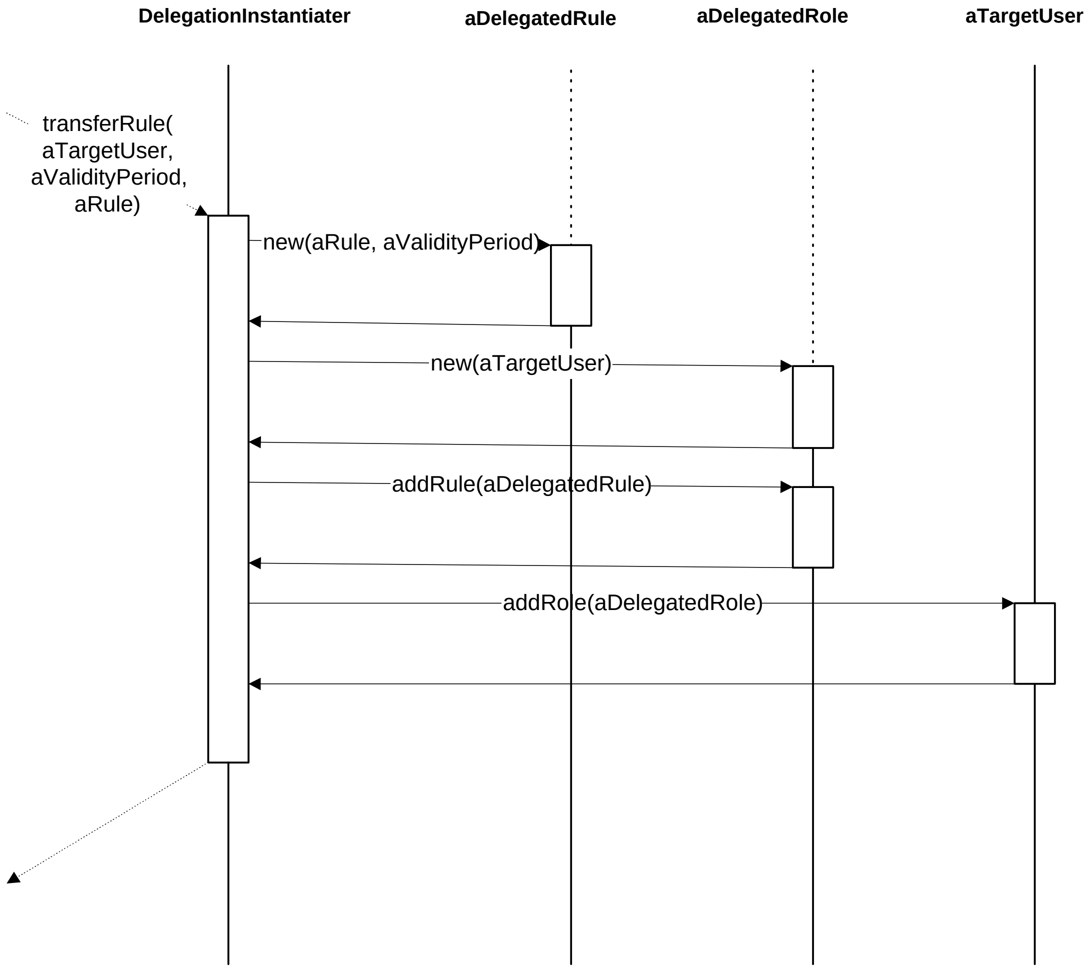
The transferRole method is called on the Delegation Instantiater to transfer a Role from a source User to a target User with a period validity constraint.

1. The DelegationInstantiater calls the getRole method on the source User.
2. The DelegationInstantiater calls the getRules method on the Role to retrieve all the Rules to transfer from the source User to the target User.
3. For each Rule to transfer, the DelegationInstantiater creates a new DelegatedRule that contains the original Rule and the validity period constraint.
4. The DelegationInstantiater creates a new DelegatedRole specific to a User (If it does not exist).
5. The DelegationInstantiater calls the addRule method on the newly created DelegatedRole to add the DelegatedRule.
6. When the DelegatedRole contains all the transferred Rules, the DelegationInstantiater calls the addRole method on the target User to add the DelegatedRole to the target User.

Scenario 2:

Transfer a Rule from a source User to a target User.

The following diagram shows the flow of requests between the objects involved when the transferRule of the DelegationInstantiater is called. The transferRule method is used to transfer, from a source User to a target User, the Access Right relative to a Property and a Class associated to a Role.



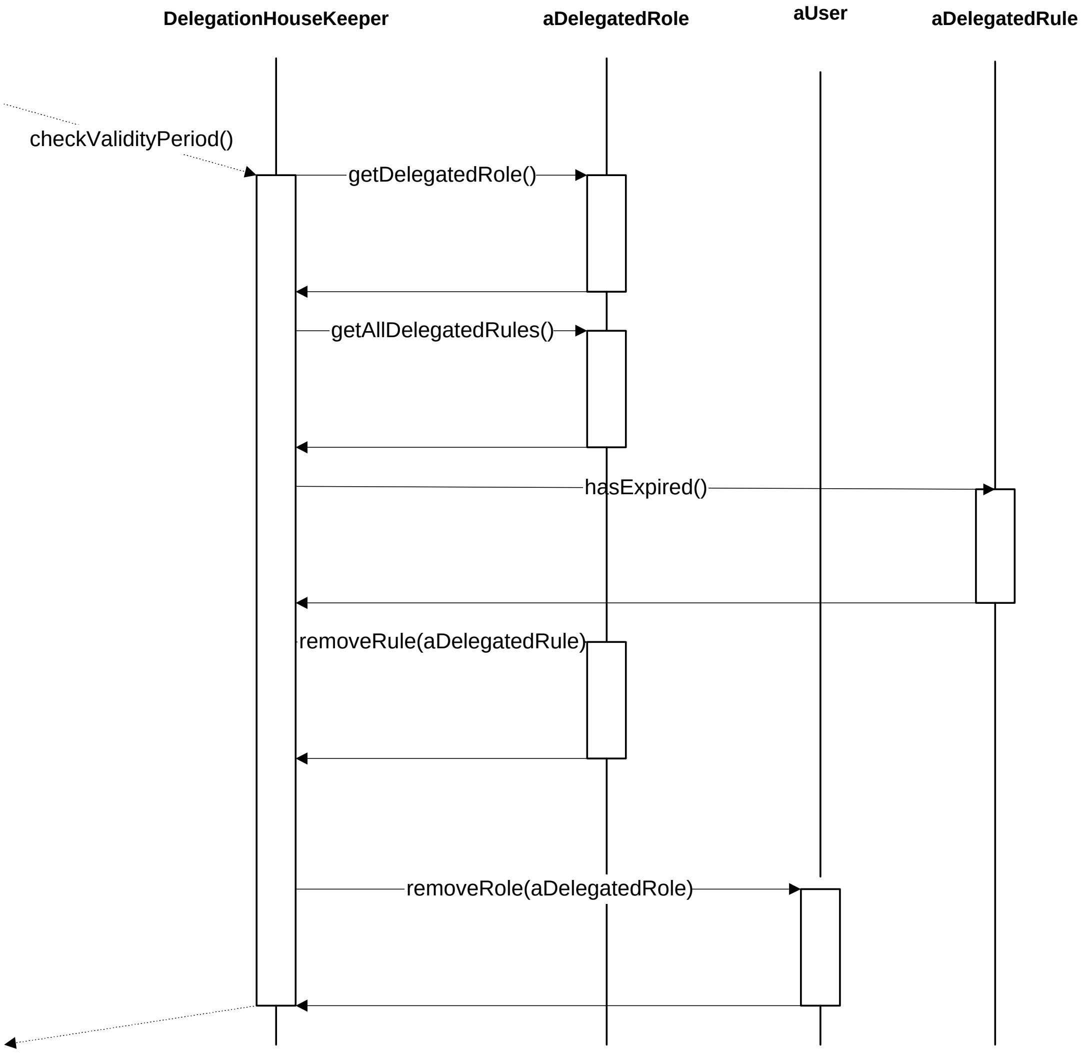
Stimuli:

The transferRule method is called on the Delegation Instantiater to transfer a Rule from a source User to a target User.

1. The DelegationInstantiater calls the getRole method on the source User.
2. The DelegationInstantiater calls the getRules method on the Role to retrieve all the Rules that are applicable for a Property or a Method of a Securable Class. Each of these returned Rules will be transferred from the source User to the target User.
3. For each Rule to transfer, the DelegationInstantiater creates a new DelegatedRule that contains the original Rule and the validity period constraint.
4. The DelagationInstantiater creates a new DelegatedRole if it didn’t exist before.
5. The DelegationInstantiater calls the addRule method on the newly created DelegatedRole to add the DelegatedRule.
6. The DelegationInstantiater calls the addRole method on the target User to add the DelegatedRole to the target User.

Scenario 3: Remove a Delegated Rule when the validity period of the Rule has expired.

The following diagram shows the flow of requests between the objects involved when a Delegated Rule is deleted because its validity period has expired.



Stimuli:

The checkValidityPeriod is called on the DelegationHouseKeeper to detect if some Delegated Rules have expired.

1. The DelegationHousekeeper calls the getAllDelegatedRoles to retrieve all the DelegatedRoles.
2. For each DelegatedRole, the DelegationHousekeeper calls the getDelegatedRules on the DelegatedRole to retrieve all the Rules that has been transferred to the User.
3. The DelegationHousekeeper calls the hasExpired method on all the returned DelegatedRules to check if they have expired.
4. If a DelegatedRule has expired, then the DelegationHousekeeeper remove the Rule from the DelegatedRole.
5. If all the DelegatedRules of a DelegatedRole have expired, then the DelegatedRole is removed from the User.

Specification

Package: com.ac.eas.security.roleDelegation

Related Frameworks

Framework: Access Control

Framework: Security Policy Management

Security Policy Management Overview

Purpose

The objective of the Security Policy Management framework is to model a solution for the specification and the management of rules to define a Security Policy. This framework provides foundation classes that other frameworks need to provide their specific security features.

This framework is used to manage:

- One or several Roles (i.e., Clerk, Manager, System Administrator, etc.)
- One or several Security Rules attached to a Role

Rules determine whether a user (i.e. his / her role(s)) is authorized to access a resource and / or whether an access must be audited. The Security Administrator is responsible for defining and managing these rules in order to reflect the Security Policy.

Services

This section describes the services provided by the Security Policy Management framework. These services are presented by describing the objects providing them.

Role

This object defines and categorizes the role(s) of a user in the system. This object should reflect the roles of a user in the enterprise. The rules defined in a Security Policy refer to these roles.

The service provided by this object in the current framework implementation focuses on the maintenance and persistence of the different roles.

It is possible to extend the concepts handled by a Role to better reflect the enterprise's organization and facilitate the administration and management of the rules. Some of these options can be:

- *Prioritizing the roles*: priorities can be associated to the different roles to represent a hierarchy of responsibilities. This could be used to derive a subset of security rules for a given user when this user has several roles (e.g. after a role delegation).
- *Adding exclusivity to a role*: an extension of the role priority can be the notion of an exclusive role. In this case, if other roles have been attached to a given user, then these roles (and the rules they maintain) are ignored
- *Defining a default role*: many rules will generally be applicable whatever role the users are. In this case, it is preferable to define a common role (e.g. "All") which is implicitly attached to all users. This facilitates the management of rules by limiting the number of rules to express.

Rule

Rules are used to express the Security Policy for the access control and the access auditing (refer to the frameworks *Access Control* and *Access Auditing*).

The resource (i.e. a property or a method) being secured or audited through the specification of rules is identified by:

Class ID

The Class Identifier of the object; the Class Identifier is defined by the object (e.g. a business object) which becomes Securable (i.e. includes access control to its implementation) and / or becomes Auditable (i.e. includes auditing to its implementation).

Operation Type

The operation type is used to denote the type of an invoked method. This is used for controlling and / or auditing the access to a property, which consists of modifying the methods to get and set the property's value. Identifying the property by its name is more intuitive, but requires an additional Operation Type key.

The three operations types are:

- 'Get': to denote a get method (i.e. an accessor) of an object's property
- 'Set' : to denote a set method (i.e. a mutator) of an object's property
- 'Invoke' : to denote any other method of the object

Resource ID

The identifier of the resource (i.e. a property name or a method name). With a 'Get' or 'Set' Operation Type, the Resource ID is the name of a property; with a 'Invoke' Operation Type, the Resource ID is the name of a method (excluding all accessor and mutator methods).

Examples of resource identification are:

- Identifying an access (read) to the property 'Balance' of the secured object 'Account'
  Class ID = 'Account' and Operation Type = 'Get' and Resource ID = 'Balance'
- Identifying an access to the method 'Transfer' of the secured object 'Account'
  Class ID = 'Account' and Operation Type = 'Invoke' and Resource ID = 'Transfer'

Every rule contains in addition the following information:

- *Message ID.* A common requirement in a security policy is to also define all messages (e.g. access violation error messages, audit messages) that are related to the resources managed through the policy. Including a message ID allows you to retrieve directly from the rule an identification of a message to display or log. The Message ID can be used and piped into the message handling system of your implementation.

- *Condition.* A rule can contain a condition to specify more complex expressions regarding the evaluation of the rule. For example, access to one property can be restricted to some roles, and only if the value of this property is less than a threshold. The expression of "less than a threshold" is handled in the condition. Rules are considered to be *complex* when they contain a condition. Rules with an empty condition (i.e. no condition) are considered to be *simple* (simply referencing the resource and the role). The expression and the evaluation of the condition are delegated and encapsulated in the object implementing the Condition interface.

The rules are specified to determine the rights of a user. The user is identified by his/her roles. The rules are expressed in relation with the roles defined in the Security Policy. A rule can be associated to several roles, and a role is attached to a set of different rules.

The rule specified above contains the following information to express security rules:

- Class ID

- Operation Type

- Resource ID

- Condition

- Message ID

- Role

This specification can be extended to meet the requirements specific to your implementation. As an example, rules used to express access auditing extend this rule (refer to Framework Access Auditing). Rules can also be extended to include constraints on their validity (refer to Framework Role Delegation).

Condition

Objects implementing the Condition interface are responsible for encapsulating the expression of a condition and managing its evaluation.

Conditions will be used when controlling the access to resources should depend on the actual values of object's properties. Thus, the Condition's expression contains references to properties. These references can be considered as parameters. The evaluation of these parameters implies retrieval of the values of properties. The Rule Management framework implementation must support a mechanism to dynamically retrieve these values. The following list provides some possible options to build this mechanism:

- Build an interpreter to convert the parameter definition into calls to the matching object's methods. This can be a rather simple task if the language used for development supports late-binding of methods (identified by strings), such as Smalltalk or Java JDK 1.1. The interpreter approach provides the best flexibility.
- Develop a custom-built late-binding mechanism if the language does not support it natively, using for example lookup tables for methods; then build an interpreter on top of this custom binding mechanism.
- Develop specialized code units associated to the evaluation of specific rules. This approach limits the flexibility and could become difficult to maintain.

In a distributed environment, it should be highlighted that the evaluation of the Condition (and embedded parameters) will perform better if local to the objects referred. It should also be easier to implement, for example when specialized code units are developed.

Also, note that every Rule contains a Condition object. Therefore, the *simple* Rules will refer to an empty Condition object, which always evaluates itself to *true*.

Rule Engine

The Rule Engine is an interface which provides general services to handle the evaluation of a set of rules. The Rule Engine object is specialized by other engine interfaces that provide the actual evaluation mechanism with respect to the type of rules they manage (refer to framework *Access Control* and *Access Auditing*).

The Rule Engine manages a set of rules associated to a given Securable or Auditable class. All the rules referring the Class ID of this Securable or Auditable are selected to populate the engine

This approach is efficient in a distributed environment where all business components (being secured and / or audited) can be located on several application servers. Indeed, centralizing the rule evaluation on one server (accessed by all other application servers) could noticeably impact the performance. It is recommended that you distribute to each component managing the secured and / or audited objects the rules associated to these objects. As a result, every component manages the security locally.

However, distributing the rules raises issues related to data synchronization. The Security Policy is commonly managed on a central Security Policy database server. Rule Engines securing objects download from this central database their rule sets and then access these sets locally.

When the Security Policy is updated on the server, it is necessary to refresh as soon as possible the rules maintained locally by the Rule Engines to preserve the policy consistency.

This data synchronization mechanism for the rules can be ensured in several ways:

- *Distributed database architecture*: Rule Engines access local databases that can be replicated or updated from the central policy server. The distributed database architecture can be implemented directly with solutions provided by DBMS software vendors.

- *Publish/subscribe service*: A Rule modified on the central policy server publishes an event. This event is handled by Rule Engines that subscribe to the service. In response to this event, Rule Engines refresh their rule sets by downloading modified rules from the central policy server. This event should not happen frequently, thus performance impact is limited. For example, the publish / subscribe mechanism implemented in the Eagle Architecture Specification can be used to ensure the refresh operation.

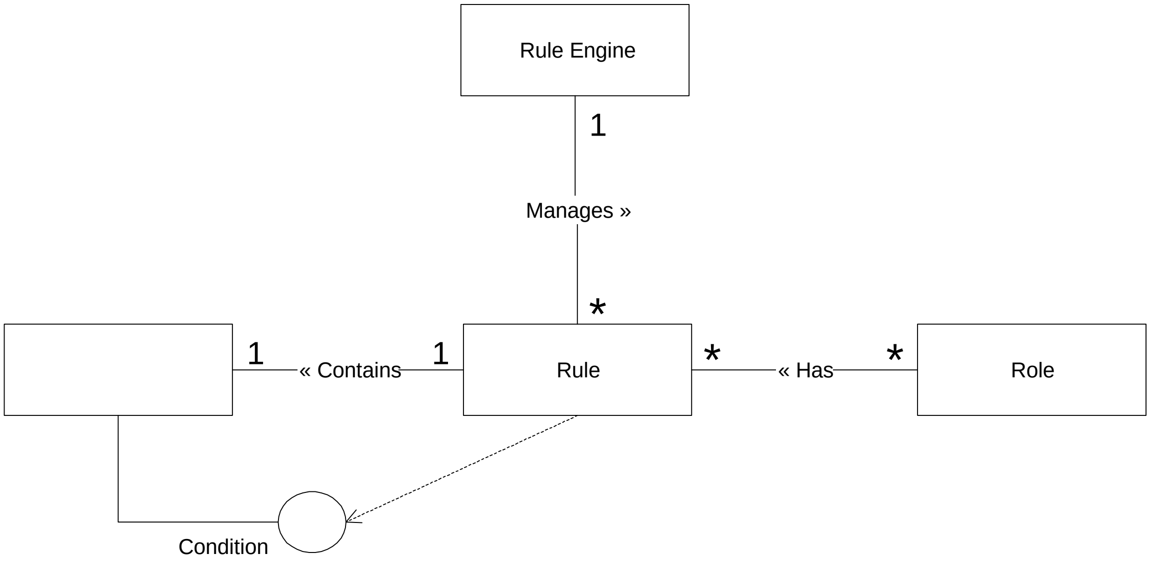
The Rule Engine populated with consistent rules is responsible to select the rule(s) that relate to the resource being accessed. When a secured and / or audited resource is accessed, the criteria used to select the rules are:

- Resource identification: the Resource ID and Operation Type, but not the Class ID as the Rule Engine is associated to one Securable or Auditable class

- Roles of the user: the user currently accessing the resource is identified by a collection of roles

Structure

Class Diagram



Participants

Role

- Identifies a role that a user can play in the system

Rule Engine

- Maintains a list of applicable rules
- Select the applicable rule(s) that satisfy the passed criteria

Rule

- Maintains the rule information
- Evaluates itself
- Triggers the evaluation of its condition
- Triggers a refresh of multiple copies when modified

Condition

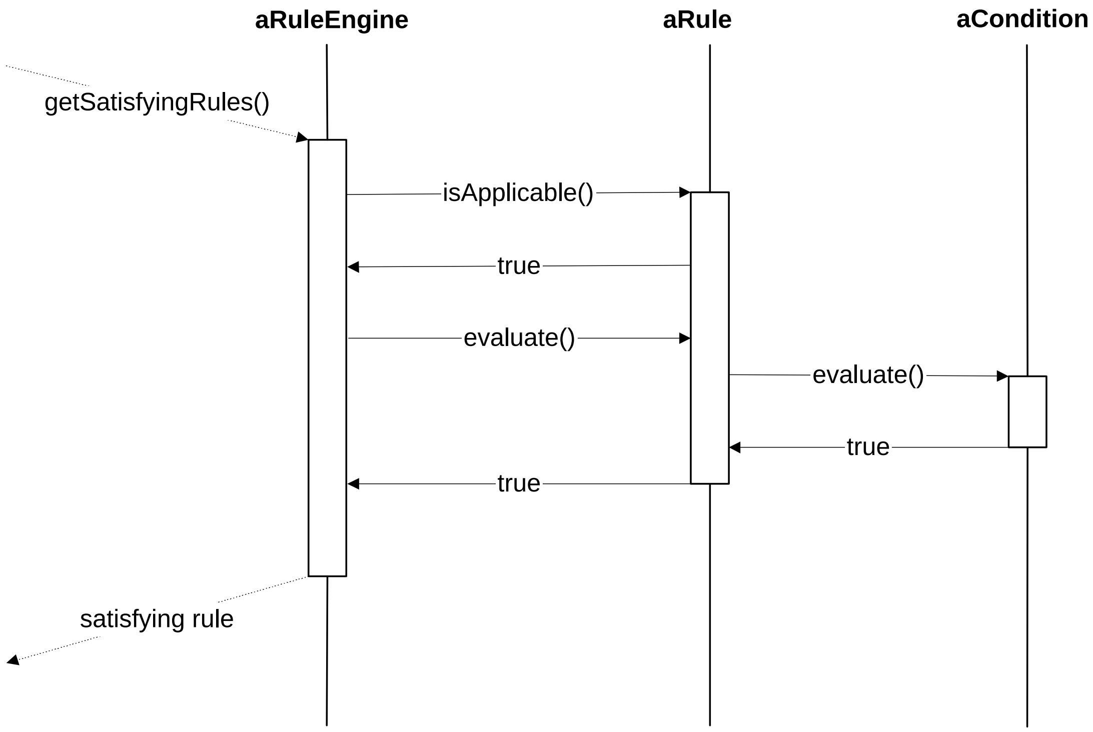
- Defines an interface for its evaluation
- Maintains the expression of the condition
- Encapsulates the evaluation of this expression
- Interfaces with mechanisms retrieving application data

Collaborations

Scenario:

A resource is accessed, the Rule Engine checks the rules it maintains to determine that the access is valid.

The following diagram shows the flow of requests between the objects involved in checking the rules maintained by a Rule Engine.



Stimuli:

A secured object's resource is accessed, the security mechanism is triggered to validate this access.

1. The Rule Engine managing the rules related to the object's Class ID is called to determine if satisfying rules exist for this access. The Rule Engine receives the following identification information:
   - Operation Type
   - Resource ID
   - Collection of roles attached to the user accessing the resource
2. The Rule Engine browses its set of rules that self determine if they match the identification criteria, i.e. are applicable
3. The Rule Engine finds an applicable Rule and triggers its evaluation
4. The Rule evaluates itself, and triggers the evaluation of its Condition
5. The Condition evaluates itself to true
6. The Rule consequently evaluates to true
7. The Rule Engine returns the Rule that has determines the access is valid

Specification

Package: com.ac.eas.security.policy

Related Frameworks

Framework: Access Auditing

Framework: Access Control

Secured UI Framework Overview

Purpose

The Security Mechanisms UI framework provides services to enforce and visualize the security policy used to control access to the Enterprise Information System. These services impact the visualization of properties and methods in two ways:

- *Proactive rights visualization*. Depending on the role of the user and the security policy, some fields can be hidden, disabled, or changed. This mechanism creates a proactive UI indication showing the operations that the user is allowed to perform. The Security Mechanisms UI framework defines services to set up a proactive user interface.
- *Reactive rights visualization*. The access rights required to perform an operation are validated when the operation is invoked. If the access rights are not sufficient, warning or error messages are raised and displayed to the user.

Design Goals

The UI framework provides a simple, flexible, and configurable layer that:

- Implements a clear separation of responsibilities between the visual representation of the Policy and the implementation of the Policy.
- Controls the display of the User Interface depending on the access rights of the user using the UI. Precisely, the Context will be used to know the Role(s) of the user.
- It is an optional extension to the User Interface Framework.

Services

Proactive Right Visualization

This Service is provided by the SecurableUserInterface as follows:

- It uses the Context returned by an object implementing the ApplicationUserManager interface to proactively check the access rights to the properties or methods of the objects implementing the Securable interface.
- According to the values defined in the Context, the object implementing the SecurableUserInterface interface changes the appearance of the graphical components displayed by the user interface.
- It associates the appearance of a graphical component with the access rights of the properties or methods of a Securable.

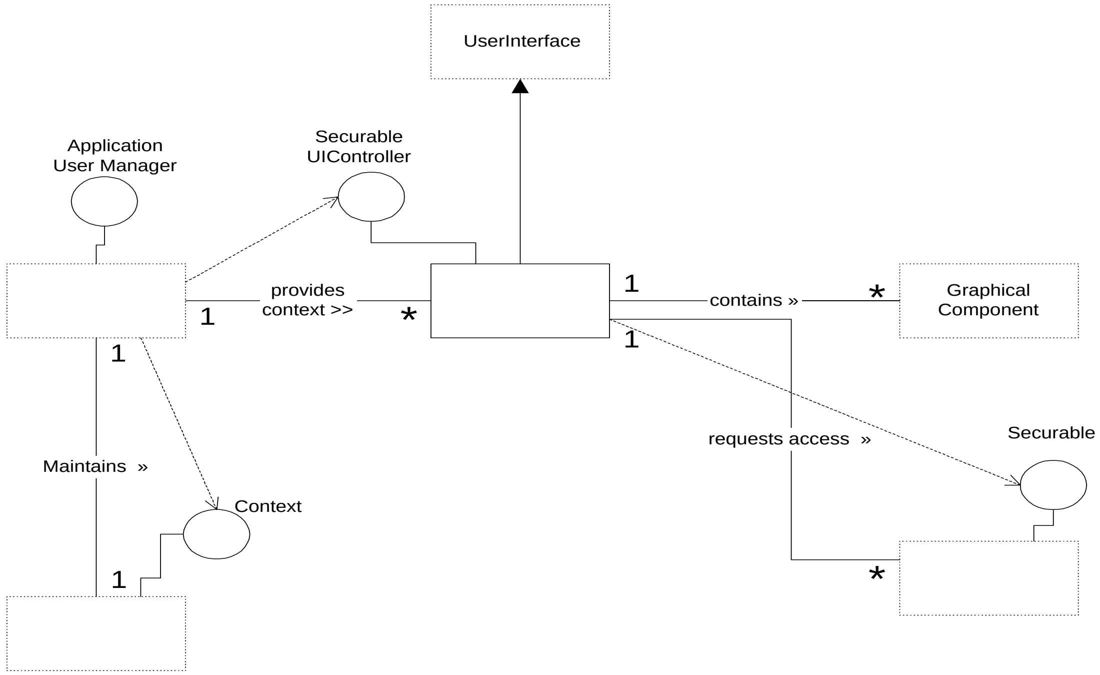
Reactive Access Control

This Service is provided by the SecurableUserInterface as follows:

- It calls the object implementing the ApplicationUserManager interface to retrieve the current Context of the user.
- It passes this Context to the objects implementing the Securable interface when the secured methods or properties of the Securable are called.
- It handles any Security Exceptions returned by the Securable objects.

Structure

Class Diagram



Participants

Graphical Component

The Graphical Components are widget class libraries responsible for handling the display of the graphical elements of a user interface. A Graphical Component must be able to hide, show, enable, or disable itself. These functions are used by the Proactive Security mechanisms to reflect the user’s access rights.

Securable

The Securable interface is implemented by objects that want to secure access to their members (Properties or Methods). See the *Access Control Framework*.

Application User Manager

The Application User Manager is responsible for providing the current UI with a Context Object corresponding to either the current user or a newly logged user. See the *Login Framework*.

Context

The Context gathers all of the information relative to the current user. See the *Login Framework*.

Securable UserInterface

The Securable UserInterface is responsible for:

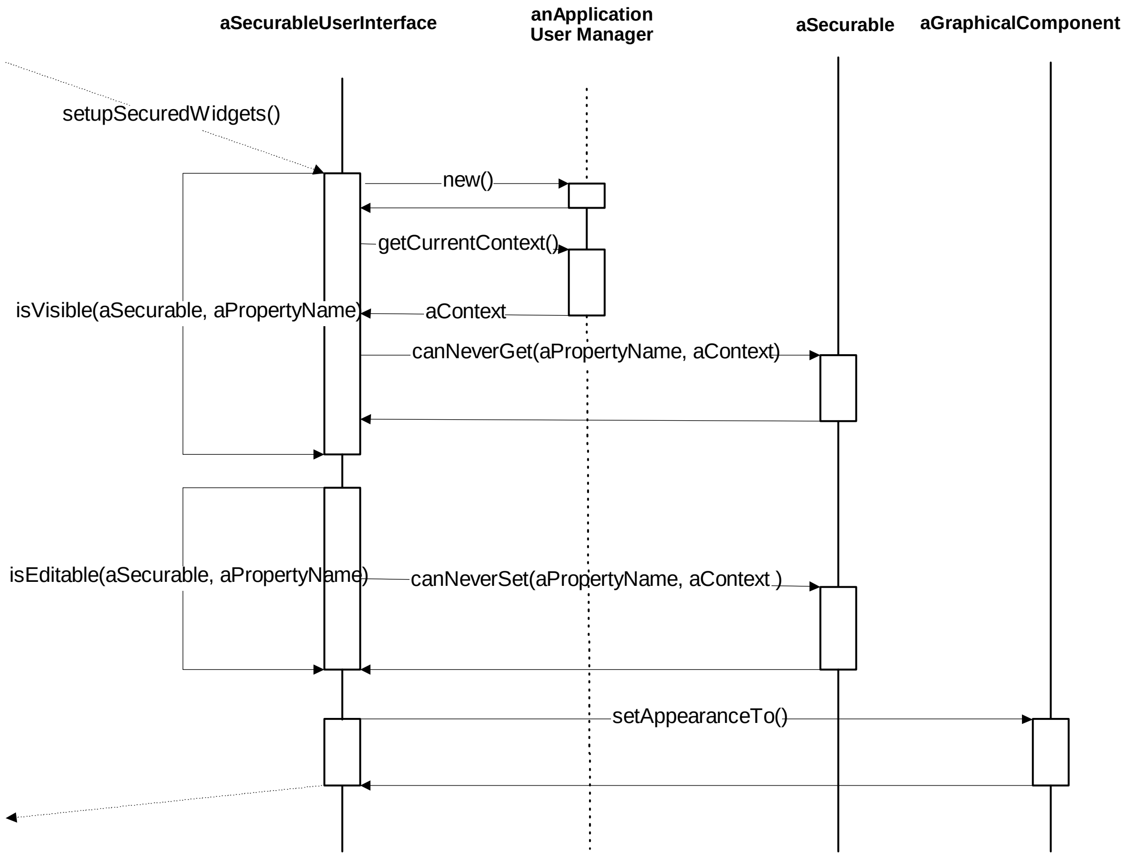
- Proactively managing the aspect of the Graphical Components to reflect the user’s access rights to Securable Objects.
- Retrieving the current Context of the user.
- Passing the Context to the Securable when calling the secured methods and properties of the Securable.
- Handling any Security Exceptions returned by the Securable.

Collaborations

Scenario 1:

Visualize access rights for a Securable's Property.

The following diagram shows the flow of requests between the objects involved when the setupSecuredWidgets of the UserInterface is called. The setupSecuredWidgets of the UserInterface is used to map the widget's behavior with the access rights of a member of a Securable Object.


aSecurableUserInterface
anApplication User Manager
aSecurable
aGraphicalComponent
setupSecuredWidgets()
new()
getCurrentContext()
isVisible(aSecurable, aPropertyName)
aContext
canNeverGet(aPropertyName, aContext)
isEditable(aSecurable, aPropertyName)
canNeverSet(aPropertyName, aContext)
setAppearanceTo()

Stimuli:

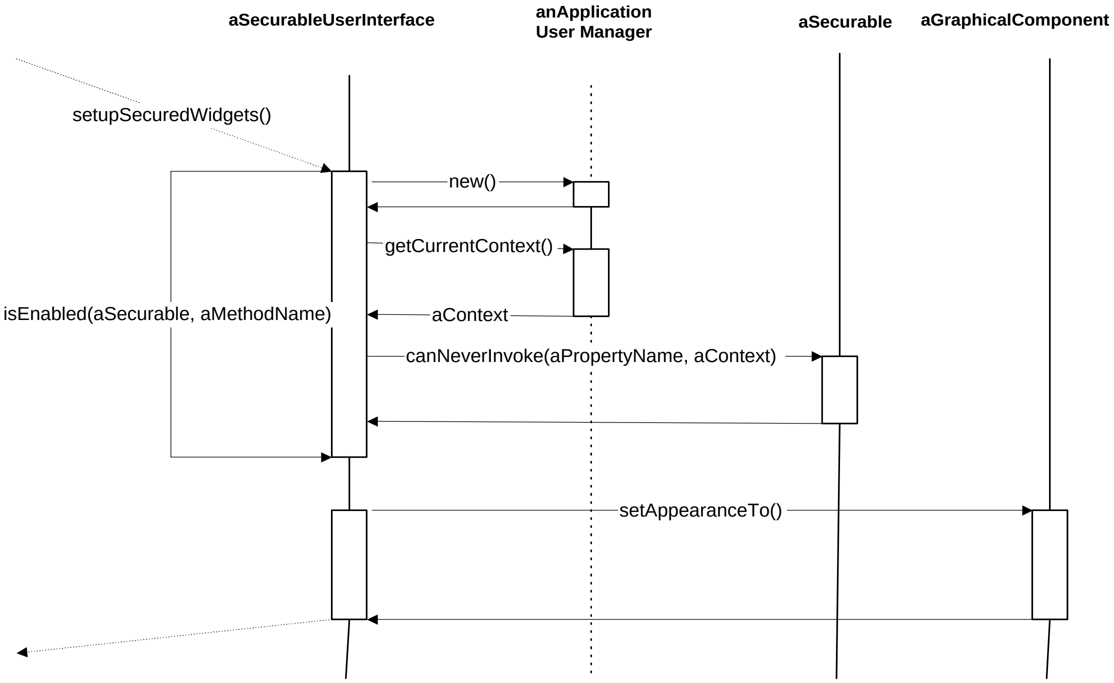
The setupSecuredWidgets method is called on a SecurableUserInterface to visualize the Access Rights of the Securable associated with the User Interface.

1. The SecurableUserInterface calls the isVisible method to check whether or not the Securable gives ‘get’ access on its property.
2. The SecurableUserInterface instantiates a new ApplicationUserManager
3. The SecurableUserInterface calls the getCurrentContext method on the Application User Manager to retrieve the Context of the current user. Once the Context is retrieved, it is cached on the SecurableUserInterface for later uses.
4. The Securable UserInterface looks at the ‘get’ access right of the Securable Property by invoking the canNeverGet method on the Securable.
5. The SecurableUserInterface calls the isEditable method to check whether or not the Securable gives ‘set’ access on its property.
6. The SecurableUserInterface looks at the ‘set’ access right of the Securable Property by invoking the canNeverSet method on the Securable.
7. According to the value returned by the isVisible and the isEditable method, the SecurableUserInterface changes the aspect of the GraphicalComponent.

Scenario 2:

Visualize access rights for a Securable's Method.

The following diagram shows the flow of requests between the objects involved when the setupSecuredWidgets of the UserInterface is called. The setupSecuredWidgets of the UserInterface is used to map the widget's behavior with the access rights of a member of a Securable Object.



Stimuli:

The setupSecuredWidgets method is called on a SecurableUserInterface to visualize the Access Rights of the Securable associated with the User Interface.

1. The SecurableUserInterface calls the isEnabled method to check whether or not the Securable gives 'invoke' access on its method.
2. The SecurableUserInterface instantiates a new ApplicationUserManager
3. The SecurableUserInterface calls the getCurrentContext method on the Context to retrieve the Context of the current user. When the Context is retrieved, it is cached on the SecurableUserInterface for later uses.
4. The SecurableUserInterface looks at the access right of the Securable Method by invoking the canNeverInvoke method on the Securable.
5. According to the value returned by this method, the Securable UserInterface changes the aspect of the Graphical Component.

Specification

Package: com.ac.eas.security.ui

Package: com.ac.eas.userinterface

Related Frameworks

Framework: Access Control

Framework: Login

Access Control Framework Overview

Purpose

The role of the Access Control framework is to secure the access to the resources of a solution. Objects compliant with the Access Control framework specifications are able to control the access to their properties and methods, and provide information about their access rights.

The Access Control framework provides support for:

- *Secure resource access*: the properties and methods of an object are accessible only to users who are entitled to access those resources
- *Secure visualization*: graphical interface components can be tailored according to the access control of the objects being displayed

The resources of an object that have to be secured must be identified, in the same way as a boundary must be defined between the public and private resources. The interface line (i.e. public vs. private) and the secure line (i.e. secured vs. non-secured) can overlap completely, partially or not at all.

"Security administrators" must specify an Access Control Policy. The Access Control Policy describes which user groups (roles) have access to secured objects' resources.

The Access Control framework is flexible both in the selection of resources to be secured and in the expression of the Access Control policy. The policy is defined by an extensible set of Access Control rules that determine if an active user shall have access to a resource. The Access Control rules can be simple rules giving direct access to a resource, or more complex and dynamic rules, including application data.

Inside an object, securing a property consists in securing the access to methods used to read (Get) and modify (Set) this property. Securing a method consists in securing access to the method execution.

Services

Overview

The Access Control framework provides three main services:

- *Securing an Object:* the object which becomes *Securable* is able to secure some of its properties and methods. This can typically be either a Business Object or an Activity.
- *Controlling the Access:* the *Access Control Manager* is responsible for determining if a resource can be accessed by the current user.
- *Specifying Access Control*: *Access Control Rules* express an Access Control Policy for all secured resources of *Securable* objects.

Securing an Object: Securable

A Securable can be any object that requires that accesses to its resource (i.e. method or property) be secured. This object must support the securable interface in order to:

- Determine if a resource access shall be granted or denied
- Handle or propagate an Access Control Violation Exception (raised by the Access Control Manager when it denies access) in its secured methods

The access violation exceptions can be managed through application-level error handling mechanisms.

The Securable object communicates with the Access Control Manager object that isolates the access control of the *get*, *set* and *invoke* operation types. The Access Control Manager limits the modifications within a Business Object to support access control.

An additional requirement for the secured methods is to accept a context that contains information about the user. This is a strong requirement due to a distributed, multi-user environment where objects can be called without having direct information about who calls them. This context allows the Securable object to provide the Access Control Manager with user information required to determine the access rights for its secured resources. The context should contain all user information required for being able to determine the access rights. This information typically includes the user’s roles, which are used to grant or deny access rights for a given user.

Controlling the Access: Access Control Manager

All Securable objects access the Access Control Manager to determine if they shall grant access to their secured resources,. Each Securable object is secured by specifying Access Control Rules for its resources (i.e. property or method). The Access Control Manager maintains Access Control Rule Engines created for each Securable object’s class.

- It responds to Securable queries to determine access control rights.
- It transparently interfaces with the rule engine associated to a Securable class; the Access Control Rule Engine infers from the Access Control rules if access to a resource shall be granted. It selects and evaluates rules that match the user’s Roles contained in the Context received by the Access Control Manager. The Access Control Rule Engine also provides an Error Message ID when it denies a resource access.

The Error Message ID returned by the Access Control Rule Engine is retrieved from the selected Access Control Rule which has denied the access. An Access Control Rule denies access when its condition is evaluated to false. Security Administrators can define error messages which detail error situations to users (see “Specifying Access Control” section).

Specifying Access Control: Access Control Rules

Access Control rules specify which resources can be accessed. Whether a resource can be accessed by a user role (such as manager, clerk, etc.) is the most common condition. If there is no Access Control rule specified for a resource, this means that the user does not have access to this resource. Access Control rules are specified for the different user roles defined in the security policy. For a given Securable object, the Access Control rules are retrieved according to the Securable class name. The Access Control rules are generally stored in a central repository (for example on a policy server), and are retrieved by the different Access Control Managers spread over the distributed environment.

An Access Control rule descriptor comprises the following information:

- Role
- Operation Type (i.e. ‘Get’ , ‘Set’ and ‘Invoke’)
- Resource ID (a property name or method name)
- Condition (if any)
- Error Message ID, a reference to the error message defined for an Access Control rule

Complex Access Control rules can be defined to include additional conditions, especially to evaluate application data; for example, "access if transaction amount < 1000". The expression of the Condition depends on the level of complexity required by the architecture implementation. It may be a single expression (operand operator operand), or a combination of several expressions linked by logical operators such as OR, AND, and NOT. This format is left open in the framework, since it depends mostly on the implementation of the framework.

The Error Message ID is used to display (and / or log) a message that explains why a resource access is denied. It is relevant to specify the Error Message ID only for complex rules. Indeed, Access Control Rules which contain no condition always grant the access to a resource, thus do not trigger an error situation. For example, a rule containing the condition ‘if transaction amount < 1000’ denies access if a user attempts to transfer more an amount greater to 1000. For that rule, the associated Message ID can refer to a message ‘You are authorized to transfer only amount < 1000.’ The message provides context sensitive feedback to users, enhancing the overall application usability.

Two types of access control information can be deduced from the set of Access Control rules:

- *Dynamic access control*: deduced from the evaluation of all access control rules (simple and complex), it results in the actual access to a Securable resource (grant or deny).

- *Static access control*: deduced when the complex rules cannot or should not be evaluated (instance not created, irrelevant data, etc.). If complex rules are expressed for a resource, the access is granted (optimistic strategy). If only simple rules are expressed for a resource, the static access control evaluates the simple rules and determine an access control right that will never change.

The dynamic access control information is used to control access within the secured resources. The static access control information can be used to obtain access right information that does not change over time. For example, a UI controller can disable a widget associated with a datum when the static access control right to update this data is denied. Mechanisms used to implement proactive vs. reactive visualization of data request this type of access control information data (refer to Framework Secured UI).

Distributing the Access Control

Distribution across the Components

In a component environment, it is preferable to implement the access control to Securable resources where Securable objects reside.

For performance considerations, all distributed Securable objects managed in one component should communicate with an Access Control Manager local to their component, rather than having to access a different component to get access control information.

Components can transparently interact with each other regardless of their physical location. Therefore intensive access to a component (if it is remote) can result in noticeable network traffic and become a potential bottleneck on low-bandwidth or overloaded networks.

Distributing the access control management along with the Securable objects results in distributing the Access Control Rules to the Securable's components.

The context received by secured methods allows you to evaluate locally the Access Control Rules, as it contains the user's roles. Passing the context all over the secured system (e.g. from client workstations to application servers hosting Securable) is acceptable as its remains small.

Distribution across the Network

Distributed object solutions often duplicate shared server object data and / or behavior to local client objects. This distribution approach is commonly used when the distributed environment suffers from network performance problems.

A common technique for supporting this distribution approach is to create Smart Proxies on the client side. A Smart Proxy caches data and / or duplicates behavior of the shared server object (hosted on an application server) in order to reduce network traffic between the distributed objects.

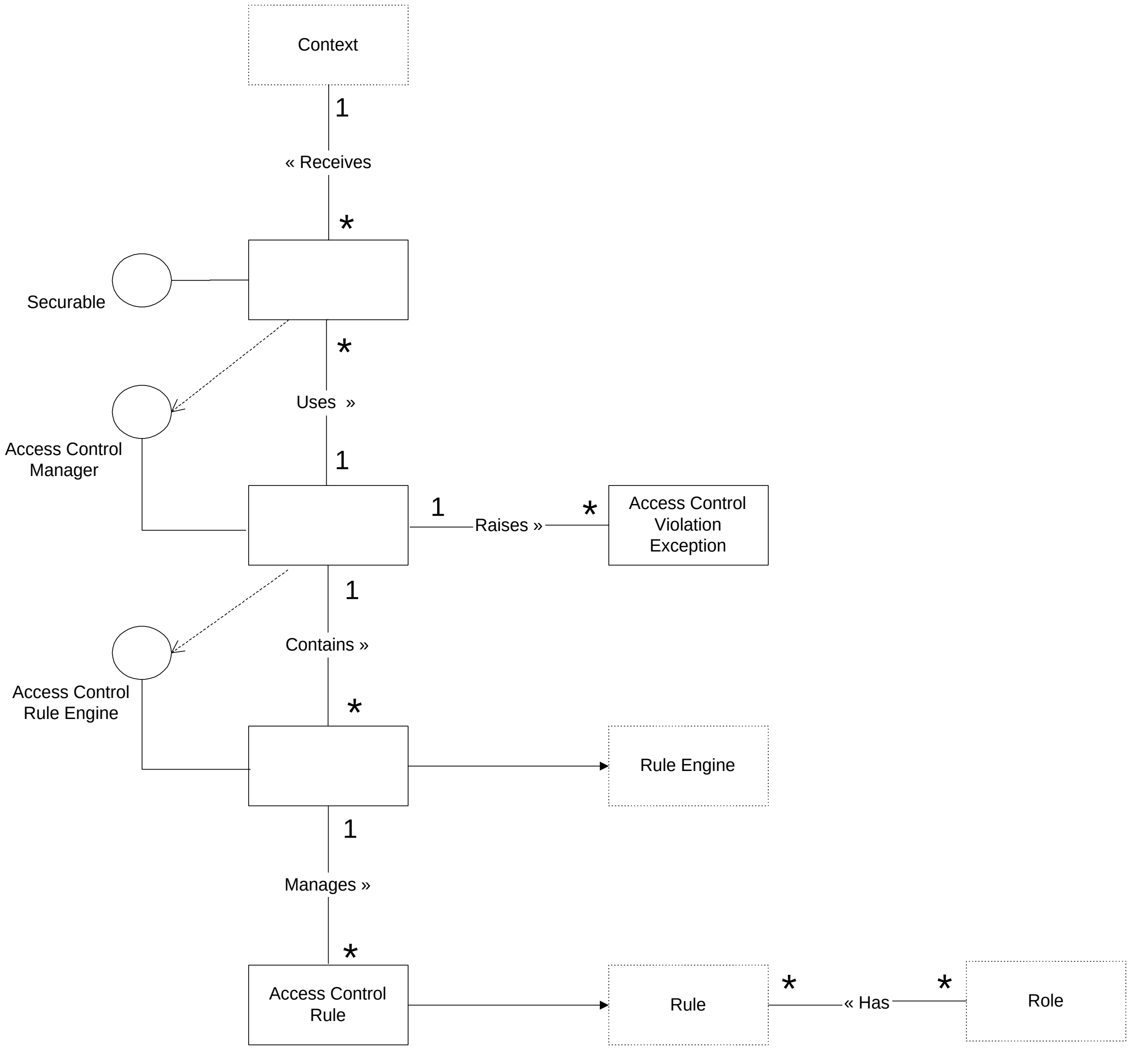
The following guidelines present different options for distributing the access control across the network depending on the required network performance and the required security flexibility:

- No performance issue
  - Distributing the Securable object is not necessary (i.e. no Smart Proxy)
  - Access Control is fully supported by the Securable hosted on the application server
  - Network traffic every time a secured resource is accessed
  - Immediate Access Control
- Low performance issue
  - Duplicating the non-secured resources (data or behavior) of the Securable objects into the Smart Proxy can limit performance penalty
  - Access control is fully supported by the Securable object hosted on the application server
  - Network traffic every time a secured resource is accessed
  - Immediate Access Control
- High performance issue
  - Duplicate all resources (i.e. not secured and secured) of the Securable into the Smart Proxy
  - The Smart Proxy is still (indirectly) secured through its Server Securable counterpart object
  - The client application can access any locally duplicated resource
  - Access control is fully supported by the Securable hosted on the application server
  - Access Control is only performed when the Smart Proxy communicates with the Securable; e.g. when the Smart Proxy is initialized with the server Securable object data, or when the Smart Proxy saves its data into the server Securable object

- High performance issue and high security flexibility
  - Requirement for immediate Access Control; for example, prevent users access to resources they are not granted rights in order to improve application responsiveness and usability
  - Duplicate all resources (i.e. not secured and secured) of the Securable into the Smart Proxy
  - Secure the Smart Proxy; this implies duplicating the Access Control Rules up to the client application (potential performance issue to perform this duplication)
  - Access Control can be performed immediately; both by the Smart Proxy and by the Securable hosted on the application server

Structure

Class Diagram



Participants

Access Control Manager

- Defines an interface used to query a resource access control
- Defines an interface used to provide objects with static access control information of secured resources
- Manages access control rule engines created for Securable classes
- Initiates the access control rule engine population
- Raises access control violation exceptions when it denies access to resources

Access Control Rule Engine

- Populates the engine with the access control rules for a given Securable object's class
- Applies the access control rules
- Provides information about the types of rules associated to a resource (complex, simple)

Access Control Rule

- Maintains the rule information
- Evaluates itself

Securable

- Implements control access in its methods that have to be secured
- Defines its Securable class name
- Propagates and / or handles access control violation exceptions

Access Control Violation Exception

- Contains the Error Message ID

Rule Engine, Rule, Role

- Refer to Framework Security Policy Management

Context

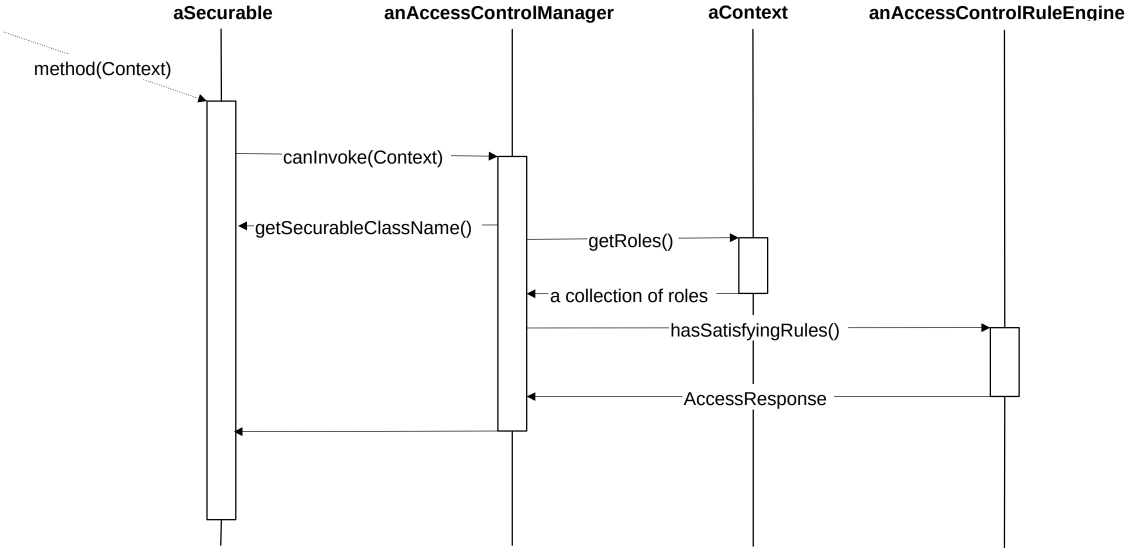
- Refer to Framework Login

Collaborations

Scenario 1:

A secured method of a securable object is invoked, the resource access is granted.

The following diagram shows the flow of requests between the objects involved in determining the access control of a securable object's secured method.



Stimuli

A secured method of a securable object is invoked.

1. The Securable calls the Access Control Manager’s method canInvoke to determine if the access is granted. The secured method receives the Context as a parameter. The Securable passes the following information:

   - Resource ID
   - Securable object
   - Context

2. The Access Control Manager retrieves the Class Name of the Securable object to lookup the Access Control Rule Engine associated to this Securable class.

3. The Access Control Manager gets the user’s roles from the Context it received. This collection of roles is used to select the applicable Access Control rules by the Access Control Rule Engine.

4. The Access Control Manager queries the Access Control Rule Engine to get the access control right for this resource, passing the following information:

   - Operation Type
   - Resource ID
   - Securable object
   - Collection of Roles

5. The Access Control Rule Engine selects the Access Control rules in relation with the Roles and evaluates them. It returns an Access Control Response to the Access Control Manager; the Access Response contains the granted access for this resource.

6. The Access Control Manager's canInvoke method analyzes the Access Response; as the access is granted, it does not raise an Access Control Violation exception.

7. The Access Control Manager returns control to the Securable object

8. The secured method in the Securable object continues its execution.

Scenario 2:

A secured property of a Securable object is accessed, the resource access is granted

Stimuli

A secured property of a Securable object is accessed (i.e. its accessor method is invoked)

The scenario is similar to Scenario 1. The objects in collaboration are the same, and the collaboration flow is identical (refer to Scenario 1 diagram). The only difference consists in the methods called to determine whether the property's access is granted or denied. For access to a property, dedicated methods (canGet, canSet) for 'Get' and 'Set' operations are called.

Using dedicated methods for 'Get,' 'Set,' and 'Invoke' operation types enhances the readability of a Securable object. An Access Control Manager deduces the operation type from the dedicated method invoked, and sends this operation type to an Access Control Rule Engine.

The following code fragment shows how a Securable object implements a secured method (it also demonstrates auditing, refer to Framework Access Auditing).

```
public void SecuredMethod(...,Context aContext)
throws AccessControlViolationException
{
    try
    {
        AccessControlManager.canInvoke("SecuredMethod",
                                       this,
                                       aContext);

        // secured method implementation
        // code executed if no access control violation exception

        // audit point (success state), if any required
        AccessAuditManager.auditInvoke("SecuredMethod",
                                       true,
                                       this,
                                       aContext);
    }
    catch (AccessControlViolationException e)
    {
        // access control violation exception raised

        // audit point (no success state), if any required
        AccessAuditManager.auditInvoke("SecuredMethod",
                                       false,
                                       this,
                                       aContext);

        // propagate the exception to the object calling
        // the secured method
        throw e;
    }
}
```

Specification

Package: com.ac.eas.security.access

Related Frameworks

Framework: Security Policy Management

Framework: Secured UI

Framework: Login

Access Auditing Framework Overview

Purpose

The Access Auditing framework provides a flexible and configurable service for auditing applications. The purpose of an audit service is to provide support for:

- *Accountability*: to be able to hold users of a system accountable for their actions within the system.
- *Security violation detection*: to be able to track attempts by unauthorized individuals to access the system, and attempts by authorized users to misuse their access to the system.

The operations to audit could be for example, a user application login or a payment transaction. They are defined by security administrators to reflect the Audit Policy. The Audit Policy is part of the overall Security Policy of a system.

Audit policies specify which resource access should be audited under what circumstances. Audit policies can be divided into two main categories:

- *System audit policies*: events related to system components such as network, servers (OS), databases (RDBMS)
- *Application audit policies*: events related to the application and business logic

The Access Auditing framework focuses on application audit. System audit is tightly coupled with product technologies and is commonly provided by software vendors.

The Access Auditing framework is flexible in the way that both the definition of which operations to audit, and the means to log them are configurable and extensible. The auditing can be controlled through a set of selection criteria to determine if an access should be logged. These selection criteria are specified within rules, which could be simple True/False facts, such as "Audit transfer if operation status is successful," or more complex and dynamic rules based on data values, such as "Audit transfer if amount > 1000." Logging can be performed using different output channels, referred to as Audit Channels. Some possible Audit Channels can be a centralized database (RDBMS), a flat file or any interface to system logging services.

The Access Auditing Service covers the following security requirements:

- *Accountability*: all audited accesses logged in audit trails contain unique user identification identifying in a secure manner the user who requested the access.
- *Privacy*: information logged should be determined according to the company’s internal security policy and/or privacy regulations.
- *Confidentiality*: all audit trails are protected and accessible only to security administrators; audit trails can be considered as confidential and being protected via access control service and/or encryption services.

The auditing process can be divided into three steps:

1. Collection (defining and collecting the resource accesses to be audited)
2. Logging (recording audited accesses into audit trails)
3. Analysis and administration (analyzing audit trails for accountability purpose or to find security holes, cleaning up audit trails, etc.)

The Access Auditing framework focuses on collecting and logging steps. The analysis and administration task is left to other components, if possible off-the-shelf solutions. It is recommended to use standard audit trail formats defined by software vendors to let analysis tools (featuring chart functions, pattern recognition, etc.) interface with those audit trails.

Services

Overview

The Access Auditing framework provides six main services:

- *Auditing an Object:* the *Auditable* is the object able to audit some of its methods. This can typically be either a Business Object or an Activity.
- *Controlling the Audit:* the *Access Audit Manager,* in coordination with the *Audit Rule Engine,* is responsible for determining if a resource access should be audited for a given user.
- *Specifying Auditing: Audit Rules* express an Audit Policy for all auditable resources of *Auditable* objects.
- *Formatting Audit Messages:* the *Audit Message Formatter* handles the formatting of audit messages with application data.
- *Managing Audit Messages:* the *Audit Message Manager* finalizes the formatting of audit messages (e.g. including user information, time stamps, etc.) and forwards those messages to *Audit Channels.*
- *Logging Audit Messages:* the *Audit Channel* manages all physical operations to write audit messages to an audit trail.

Auditing an Object: Auditable

An Auditable can be any object that requires that the accesses to its resources (i.e. method or property) be audited. This object must support the Auditable interface in order to determine if a resource access should be audited.

The Auditable object communicates with an Access Audit Manager object that isolates auditing of the get, set, and invoke operation types. The Access Audit Manager limits the modifications within a Business Object to support auditing. It can be adapted without modifying Auditable objects.

The same way as a boundary must be defined between public and private resources, a boundary must be defined between the resources that are to be auditable and those that are not. The interface line (i.e. Public vs. Private) and the auditing line (i.e. Auditable vs. Non Auditable) can overlap completely, partially, or not at all.

An additional requirement for auditable methods is to receive a context that contains information about the user. This strong requirement is due to a distributed, multi-user environment where objects can be called without having direct information about who calls them. This context allows the auditable object to provide the Access Audit Manager with user information required to determine the access audit rights for its auditable resources.

The context contains all information required for determining locally the access audit rights. This information basically consists in a list of the roles for a given user. These roles are used to authorize an access audit based on the Audit Rules specified for an Auditable object. Receiving user's information directly in the auditable resources eliminates additional accesses to distributed / remote components managing user information. The multiplication of distributed accesses can have a noticeable performance impact and shall be limited.

Controlling the Audit: Access Audit Manager

All auditable objects communicate with the Access Audit Manager to determine if they shall audit a resource.

- It responds to an auditable object's queries to determine if a property's*get* or *set* operation, or a method invocation shall be audited.
- It transparently interfaces with the rule engine for the evaluation of the Audit Rules; the Audit Rule Engine infers from the Audit rules if a resource access should be audited. It selects and evaluates rules that match the user's Roles contained in the Context passed by the Access Audit Manager. The Audit Rule Engine also provides an Audit Message ID when it determines an audit is needed.

The Audit Message ID returned by the Audit Rule Engine is retrieved from the Audit Rule which confirms a need to audit the resource. Security Administrators are responsible for defining these audit messages.

Determining the audit status of Auditable resources should be a local operation within the same component as managing those Auditable objects. For performance considerations in a distributed environment, it is preferable that all Auditable objects in a given component use an Access Audit Manager local to this. As a result, Auditable objects do not need to continuously access remote components (potential performance bottleneck) responsible for managing access auditing. The Audit Rules are distributed across the environment wherever Auditable objects are.

Specifying Auditing: Audit Rules

The Access Auditing framework controls the audit using a set of Audit Rules to specify which resource accesses to audit.

- They maintain the information about which resources to audit, under what conditions
- They evaluate their condition (complex rules)

The specification of audit rules is close to the specification used for rules defining Access Control rights (refer to Framework Access Control). The only difference is that the former are used to require an auditing while the latter are used to deny an access. Audit rules are specified for the different user roles defined in the security policy. For a given Auditable object, the audit rules are retrieved according to the Auditable class name. The Audit Rules are generally stored in a central repository (for example on a policy server), and are retrieved by the Access Audit Managers integrated into the components of a distributed environment.

The completion status of an operation is the most common condition used to determine if an audit message should be recorded. For example, a successful access to a resource may not be audited, but a failed access to the same resource must be audited. Most audit rules are going to be defined to specify this type of condition. If a rule does not exist for an operation triggered on an auditable object, then it is not logged. This ensures that audit trails reflect audit policies defined by security administrators.

An audit rule descriptor comprises the following information:

- Role
- Operation Type, i.e. "Get," "Set," and "Invoke"
- Resource ID (a property name or method name) to be audited
- Operation Status ("Success," "No Success")
- Condition (if any)
- Message ID, a reference to the audit message defined for an audit rule

Complex audit rules can be defined to include additional conditions, especially to evaluate application data; for example, "audit if transaction amount > 1000." The expression of the Condition depends on the level of complexity required by the architecture implementation. It may be a single expression (operand operator operand), or a combination of several expressions linked by logical operators such as OR, AND, and NOT. This format is left open in the framework, since it depends mostly on the implementation of the framework.

Integrating the Access Auditing into the components where Auditable objects are managed presents several advantages for the evaluation of complex rules:

- No performance hit resulting from the transfer of application data used in the conditions of complex rules to a distributed / remote component managing the Access Auditing
- More flexible implementation of a condition evaluator, as objects referenced in complex rules can be accessed locally

Formatting Audit Messages: Audit Message Formatter

The Audit Message Formatter is responsible for formatting audit messages that can contain references to application data. These references, called parameters, require accessing the Auditable object's property values that are to be replaced with actual data.

- It handles the formatting of an Audit Message identified by an audit message ID
- It retrieves the Audit Message, including a message text

As for the evaluation of audit rules, it is preferable to have a local message formatter within the component managing one or more Auditable objects. This avoids any access to a remote formatting service in a distributed environment (which can penalize the performance), and simplify the implementation of a Message Formatter (objects can be accessed locally).

Managing Audit Messages: Audit Message Manager

The Audit Message Manager is the entry point used by Auditable objects that have identified a need for logging an audit message. The Audit Message Manager coordinates the processing and logging of audit messages.

- It requests the audit message formatting service to format the audit message; the Message Formatter handles the formatting of an Audit Message identified by an audit message ID. The Message Formatter can also provide the Audit Message Manager with a message severity to let it select severity-based audit channels.
- It generates a complete audit record and sends it to one or more audit channels.
- It manages a list of possible (available) audit channels
- It can manage a selection mechanism to determine which audit channel to use according to the severity of the audit message.

The definition of audit messages is the responsibility of security administrators. Audit messages shall be stored in an external catalog to facilitate localization.

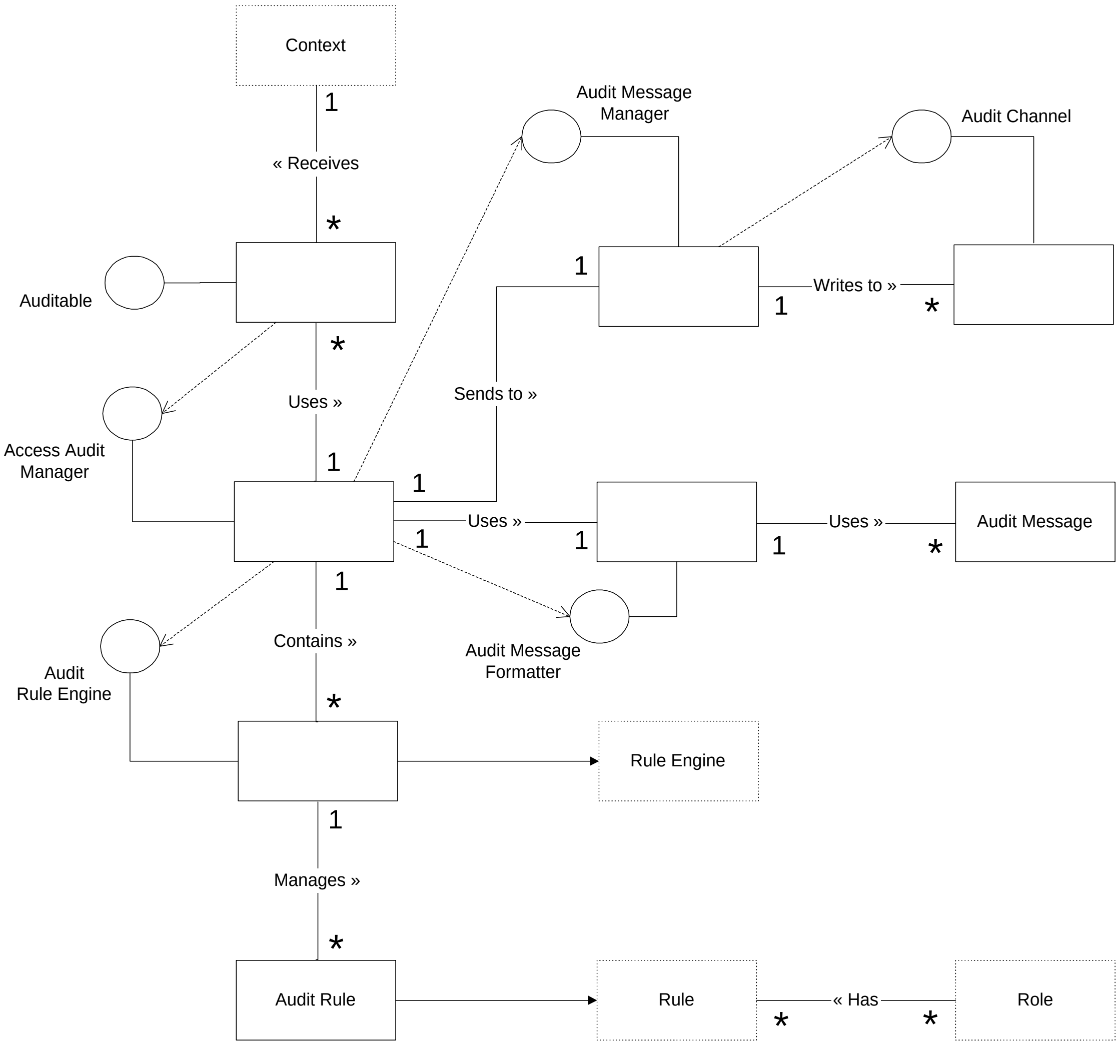
Logging Audit Messages: Audit Channel

The Audit Channel is responsible for the physical logging of audit messages (writing to a database, a file, etc.). The Audit Channel can be a custom, built-in component for a simple application environment, a component offering its services over the network in a distributed environment, or even be an interface to a standard software product for audit trail management.

It is recommended that you implement a mechanism providing fail-over for audit channels. This can be a critical system management requirement in a distributed environment where several applications may use the same audit channel service on the network. In that case, a single point of failure on the audit channel can halt several systems.

Structure

Class Diagram



Participants

Access Audit Manager

- Defines methods to query the need for auditing a resource access
- Manages audit rule engines created for Auditable classes
- Initiates the audit control rule engine population
- Returns an audit response containing a status and/or a message ID

Audit Rule Engine

- Applies the audit rules
- Populates the engine with the audit rules for a given Auditable object's class
- Retrieves the Audit Message ID associated with an audit rule

Audit Rule

- Maintains the rule information
- Evaluates itself

Auditable

- Defines an interface for objects that have auditable operations and properties
- Defines its Auditable class name
- Implements audit points in its methods that have to be audited

Audit Message Formatter

- Formats an Audit Message, replacing parameters with actual data values of the Auditable object
- Retrieves the Audit Message identified by an Audit Message ID
- Returns to the Access Audit Manager the formatted audit message

Audit Message

- Includes an Audit Message string and/or severity

Audit Message Manager

- Sends to the audit channel(s) the audit information to be logged
- Performs the final creation of an audit record, which contains not only the formatted message, but also information such as a time stamp, an user ID
- Manages a list of audit channels distinguished by severity level
- Handles the selection of an audit channel according to the audit message severity (if supported)

Audit Channel

- Manages an audit trail
- Writes to the audit trail

Rule Engine, Rule, Role

- Refer to Framework Security Policy Management

Context

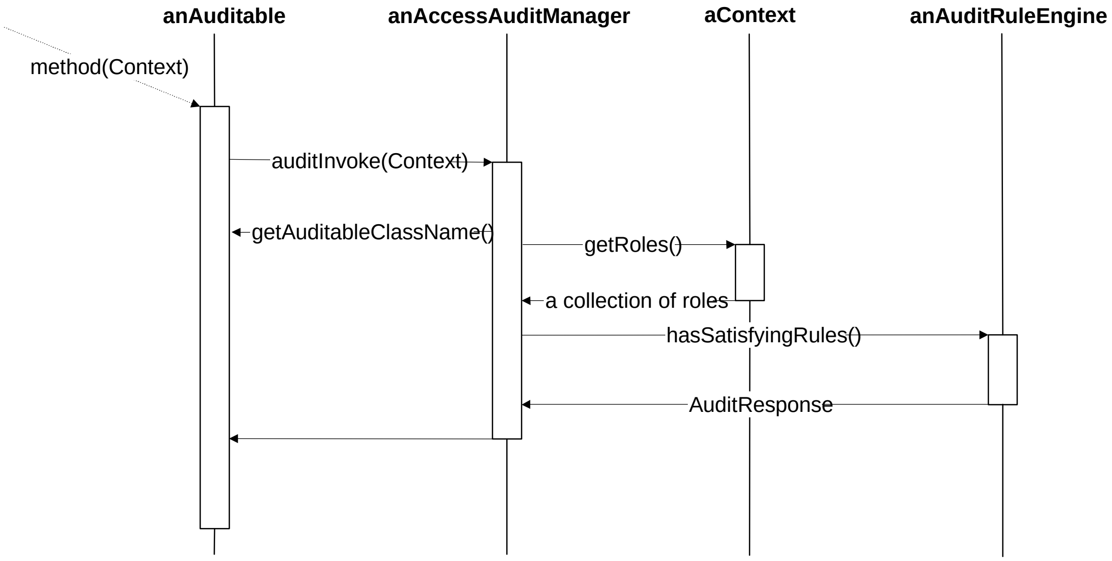
- Refer to Framework Login

Collaborations

Scenario 1:

An audited method of an auditable object is invoked by another object, the need to audit the resource access is determined.

The following diagram shows the flow of requests between the objects involved in determining the access audit of an auditable object's audited method.



Stimuli:

An audited method of an auditable object is invoked on the object.

1. The Auditable calls the Access Audit Manager’s method auditInvoke to process the audit of this method access. The Auditable passes the following information:

    - Resource ID
    - Operation Status
    - Auditable object
    - Context

2. The Access Audit Manager retrieves the Class Name of the Auditable object, to lookup the Audit Rule Engine associated to this Auditable class.

3. The Access Audit Manager gets the user’s roles from the Context it received. This collection of roles is used to select the applicable Audit rules by the Audit Rule Engine.

4. The Access Audit Manager queries the Audit Rule Engine to know if it shall audit this resource access, passing the following information:

    - Operation Type (i.e. ‘Invoke’)
    - Resource ID
    - Operation Status
    - Auditable object
    - Collection of Roles

5. The Audit Rule Engine selects the Audit rules in relation with the Roles, and evaluates them. It returns an Audit Response to the Access Audit Manager. The response contains:

    - Audit status (to audit or not to audit)
    - Audit message ID (null if audit not needed)

6. The Access Audit Manager's auditInvoke method analyzes the Audit Response and processes the audit according to the audit status contained in the response (i.e. to audit or not to audit)

7. The Auditable's audited method continues its execution.

Scenario 2:

A property of an auditable object is accessed by another object, the need to audit the resource access is determined.

Stimuli

A property of an auditable object is accessed (i.e. its accessor method is invoked)

This scenario is similar to Scenario 1. The objects in collaboration are the same, and the flow between them is identical (refer to Scenario 1 diagram).

The only difference resides in the methods used to check the need to audit. For access to a property, dedicated methods (auditGet, auditSet) for ‘Get’ and ‘Set’ are called.

The usage of specific methods for ‘Get,’ ‘Set,’ and ‘Invoke’ operation types enhances the readability of an Auditable object. An Access Audit Manager deduces the operation type from the dedicated method, and sends this operation type to an Audit Rule Engine.

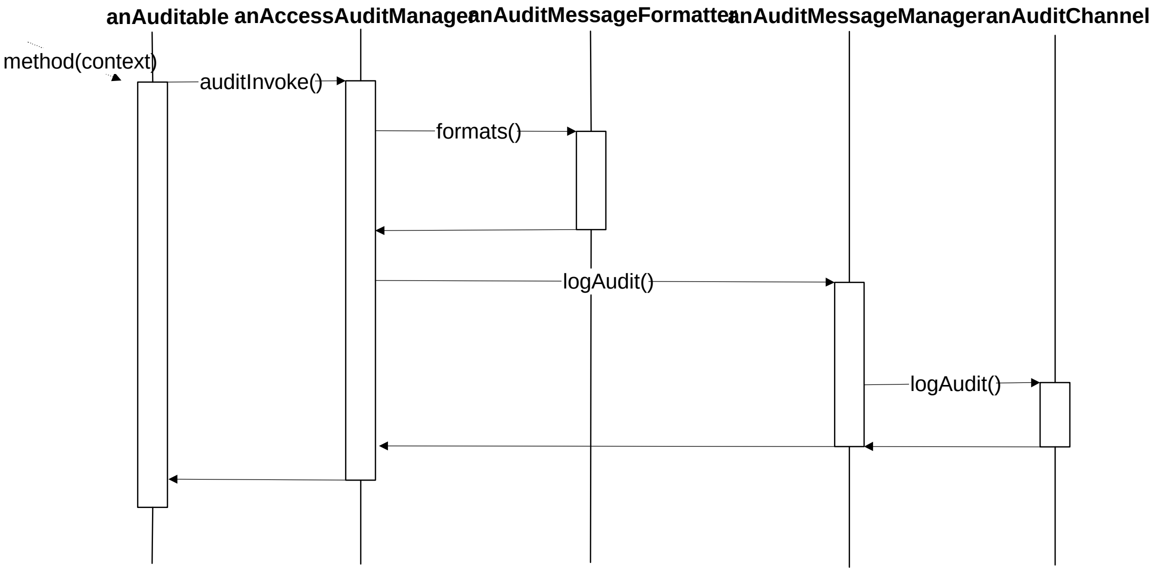
The following code fragment shows how an Auditable object implements an audited method (it also demonstrates access control, refer to Framework Access Control).

```
public void getProperty(Context aContext)
throws AccessControlViolationException
{
    try
    {
        AccessControlManager.canGet("Property",
                                    this,
                                    aContext);

        // code executed if no access control violation exception

        // audit point where completion of the operation is
        // successful(i.e. true flag for success state)

        AccessAuditManager.auditGet("Property",
                                    true,
                                    this,
                                    aContext);

        return Property;
    }
    catch (AccessControlViolationException e)
    {
        // access control violation exception raised

        // audit point where completion of the operation is
        // not successful(i.e. false flag for no success state)

        AccessAuditManager.auditGet("Property",
                                    false,
                                    this, aContext);

        // propagate the exception to the object calling
        // the secured method
        throw e;
    }
}
```

Scenario 3:

Log the resource access audit for an audited method.

The following diagram shows the flow of requests between the objects involved when an auditable object audits one method declared to be auditable



Stimuli:

An auditable method of an auditable object is invoked on the object; the resource has to be audited

1. The Access Audit Manager determines when its auditInvoke method is called that the resource access shall be audited (refer to Scenario 1 and Scenario 2 for determining the need to audit). It has received an Audit Response from an Audit Rule Engine which contains an Audit Message ID
2. The Access Audit Manager makes a request to the Audit Message Formatter to format the Audit Message identified by the given message ID, passing the following information:
   - Message ID
   - Auditable object
3. The Audit Message Formatter retrieves the Audit Message from a message catalog (using the ID), and formats the audit string by replacing parameters with actual data values of the Auditable object
4. The Audit Message Formatter returns the formatted audit message to the Access Audit Manager.

5. The Access Audit Manager forwards the formatted audit message to the Audit Message Manager, passing the following information
   - Audit Message (including the formatted audit string)
   - Context
6. The Audit Message Manager appends to the audit string (received in the Audit Message) information such as a time stamp, the user's principal ID (retrieved from the Context passed)
7. The Audit Message Manager sends the final message to one or more Audit Channels selected according to the Audit Message's severity
8. The Audit Channel logs the message into an audit trail and returns to the Audit Message Manager
9. The Audit Message Manager returns to the Access Audit Manager
10. The Access Audit Manager has finished processing the audit and returns control to the Auditable
11. The Auditable's audited method continues its execution

Specification

Package: com.ac.eas.security.audit

Related Frameworks

Framework: Security Policy Management

Framework: Access Control

Framework: Login

Non-Repudiation Framework Overview

Purpose

The four basic cryptographic services are confidentiality, authentication, data-integrity, and non-repudiation. To establish a secure communication between several systems, all of these services need to be provided. There are several ways to do this:

- Cryptographic services can be delivered at the transport level using technologies such as SSL. This provides the services *on-the-fly* for each piece of information sent, for example, a TCP/IP packet.
- The services can be provided at the application level, which means that an application must be enabled to encrypt/decrypt, authenticate, and add controls to the data before transmitting it.

These approaches are not fully equivalent with regard to efficiency, flexibility, complexity, and security strength. Considering these factors and the technologies available, confidentiality, authentication, and data-integrity services are usually delivered at the transport level.

It makes more sense to associate non-repudiation with business operations, which are performed at the application level. In a traditional business transaction, paper receipts definitively establish the exchange of funds or goods. These receipts can be stored and later exhibited to settle disputes about the conditions of a transaction. The Non-Repudiation framework aims to provide an equivalent to paper receipts for an electronic transaction.

Using non-repudiation, parties in a transaction are protected from false claims about the status or occurrence of a transaction. This ensures that parties involved in a transaction cannot claim that it never occurred. To achieve such a service, we need to provide:

- A way to specify which of the services delivered by business objects need to be non-repudiable, and how these business objects can deliver non-repudiable receipts in an understandable, human-readable fashion.
- A way to deliver non-repudiation services to the objects in an open and encapsulated way. The Non Repudiation framework provides a way of accepting various non-repudiation systems with a common interface. Non repudiation systems can be added, removed, or upgraded without requiring any changes to other services. For example, an application developer can request a specific service without any knowledge of how the service is implemented.
- A way to provide proof of origin and receipt and to be able to exhibit them if needed. This framework does not describe how to do this, but various possibilities can be considered, for example, database storage, audit service, or flat files.

Services

The Non-Repudiation framework provides complete non-repudiation for a transaction. Complete non-repudiation involves:

- *Non-repudiation of origin*: when a business object requests a service, the request is signed in an unforgeable way by the requester who then transmits it to the service provider. This can be exhibited later to prove that the request existed.
- *Non-repudiation of reception*: when processing a request, the service provider sends a signed receipt back to the service requester. This can be used later to prove that the provider agreed to deliver the service.

Request

The Request object represents the nature of the transaction; that is, the kind of services that are requested by one business object to another.

This object may only be a conceptual object and may not be associated to a class in an implementation. Typically, if a request from a service consists of invoking a remote method on a component, there is no object associated to the request, only a method call.

Receipt

The Receipt object represents the delivery of a service by a service provider to another business object.

Like the Request object, the Receipt may only be a conceptual object and may not be associated to a class in an implementation. Typically, if the request consists of invoking a remote method on a component, the delivery of the service can be a return value and/or a modification of the "out" parameters.

The Request and Receipt objects both benefit from the non-repudiation capabilities inherited from the object, NonRepudiable.

Non-repudiable

The NonRepudiable object contains the information that provides non-repudiation to Request and Receipt objects. This information must be certified by an unforgeable signature. The NonRepudiable object knows how to obtain unforgeable signatures and knows how to check them. It may also provide methods to be persisted if Request or Receipt are not able to persist themselves.

Non-repudiation Adapter

This object is the engine used to deliver unforgeable signatures using cryptographic algorithms such as the RSA Digital Signature. The Adapter pattern (c.f. Design Patterns - Addison Wesley) is used in the design of this engine to encapsulate the implementation of the non-repudiation mechanism. Using Adapter, it is possible to choose between several non-repudiation mechanisms and to plug in new ones easily, even if every mechanism has a different interface.

Typically, a NonRepudiationAdapter uses a private key to produce a NonRepudiableSignature. This key is an identifier for a unique entity. This fact implies, for security reasons, that Adapters have to be local to each principal; they cannot be a CORBA service. So, if two systems want to benefit from non-repudiation when communicating with each other, they should implement the same non-repudiation mechanisms.

Non-repudiable Signature

This object is the cipher data corresponding to the information maintained by the NonRepudiable. It is delivered by a NonRepudiationAdapter and should provide information on which a non-repudiation mechanism has been used.

Persistence Strategy

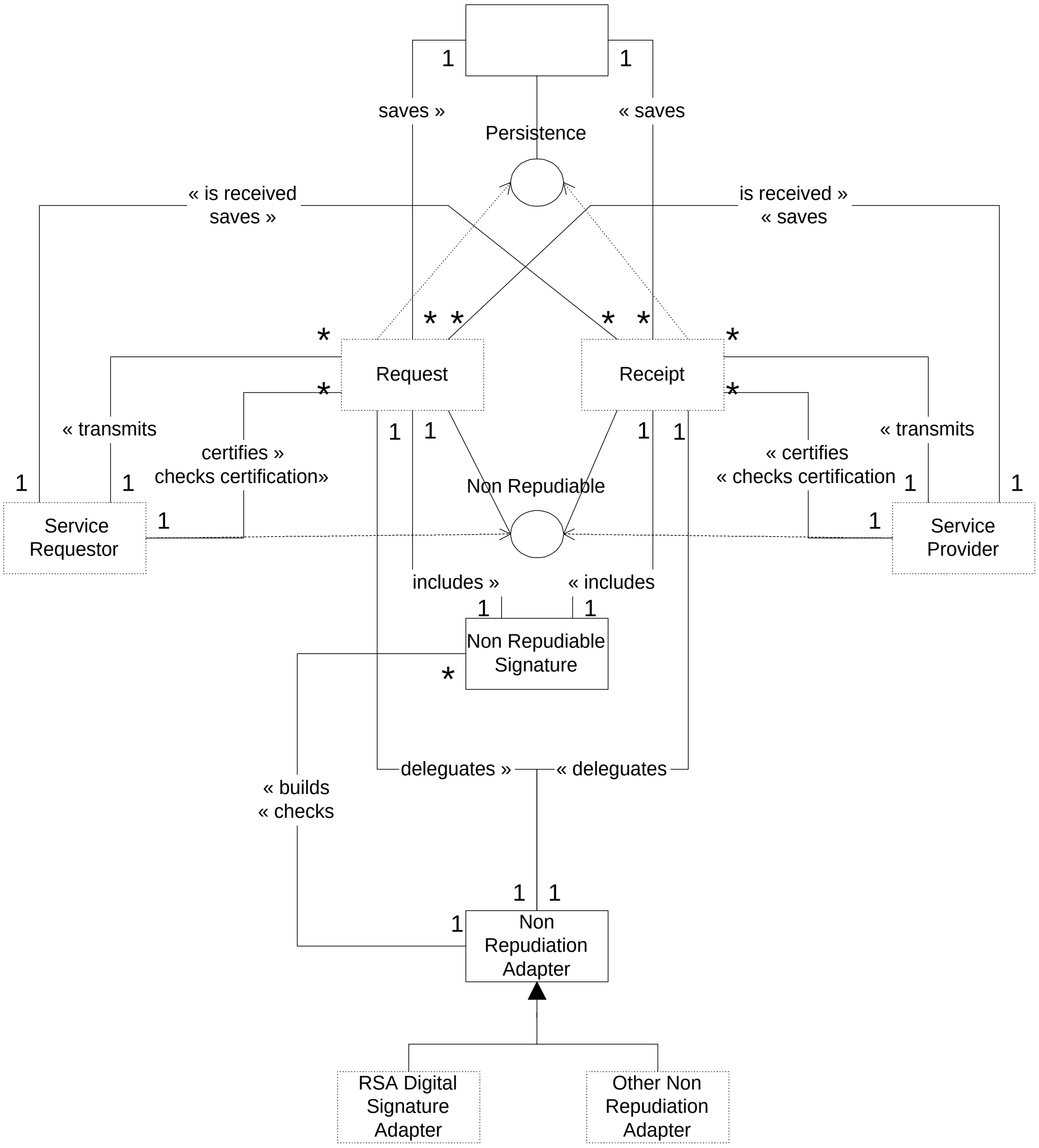
Parties involved in a transaction may want to later exhibit the non-repudiation certificates that they received when the transaction occurred. Since this is the purpose of the *Persistence Framework*, this framework does not describe how to achieve such a persistence, but several ways can be suggested:

- Auditing feature (see Access Auditing framework)
- Persistence in database (see *Persistence framework*)
- Flat file storage

The use of a Strategy Pattern in the design of this service would allow the transparent use of several of the methods listed above, as well as other available mechanisms.

Structure

Class Diagram



Participants

Request

A Request represents a formatted demand for a service from one object to another.

Receipt

A Receipt represents a formatted acknowledgement related to the reception of a request.

ServiceRequestor

Any application object that requests a specific service from a provider in a non-repudiable fashion:

- The ServiceRequestor formats a non-repudiable Request.
- The ServiceRequestor transmits a non-repudiable Request to a ServiceProvider.
- The ServiceRequestor asks the ServiceProvider for a non-repudiable Receipt.
- The ServiceRequestor checks the validity of the certification of the Receipt.
- The ServiceRequestor saves and retrieves the Receipt.

ServiceProvider

Any business object that provides a specific service to a requestor in a non-repudiable fashion.

- The ServiceProvider checks Requests sent by a ServiceRequestor.
- The ServiceRequestor saves and retrieves the non-repudiable Request.
- The ServiceProvider processes Requests sent by a ServiceRequestor.
- The ServiceProvider formats a non-repudiable Receipt.
- The ServiceProvider transmits a non-repudiable Receipt to the ServiceRequestor.

NonRepudiable

Manages the information that provides non-repudiation to Requests and Receipts.

- The NonRepudiable obtains and maintains a NonRepudiableSignature.
- The NonRepudiable checks its NonRepudiableSignature.
- The NonRepudiable may have to persist itself.

NonRepudiableSignature

Contains data to enable non-repudiation.

- Cipher data certifying the validity of the NonRepudiable.
- Identification of the non-repudiation mechanism used.

NonRepudiableAdapter

Intermediary between various non-repudiation mechanisms and NonRepudiable.

- The NonRepudiableAdapter builds a NonRepudiableSignature.
- The NonRepudiableAdapter checks NonRepudiableSignatures built by the same kind of Adapter (adapters for the same non-repudiation mechanism).

The use of the Adapter Pattern enables the use of various non-repudiation techniques. This approach is fully compatible with the JDK1.1 engines defined in the Java Cryptography Extension.

PersistenceStrategy

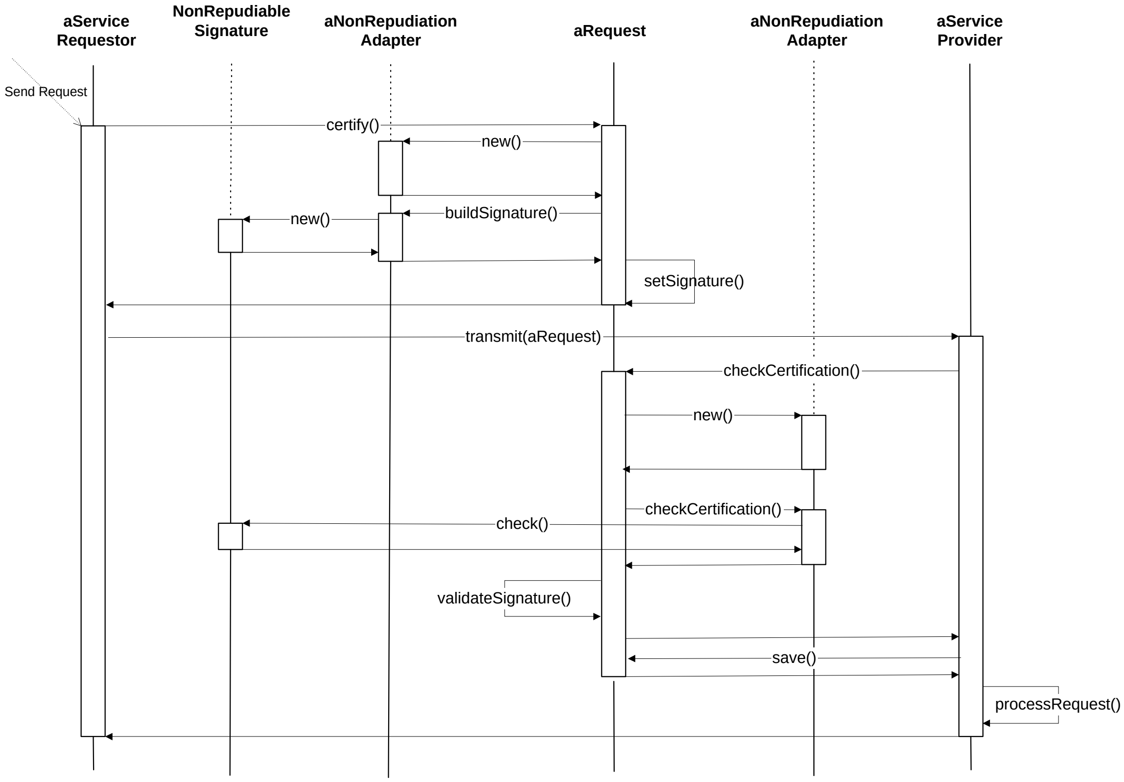
Allows the persistence of Request and Receipts. The use of a Strategy Pattern for the design of this object may allow it to transparently enable the use of several persistence mechanisms.

Collaborations

Scenario:

A ServiceRequestor asks a ServiceProvider to process a request in a non-repudiable fashion.

The following diagram shows the flow of calls needed to provide non-repudiation to a Request.



Stimuli:

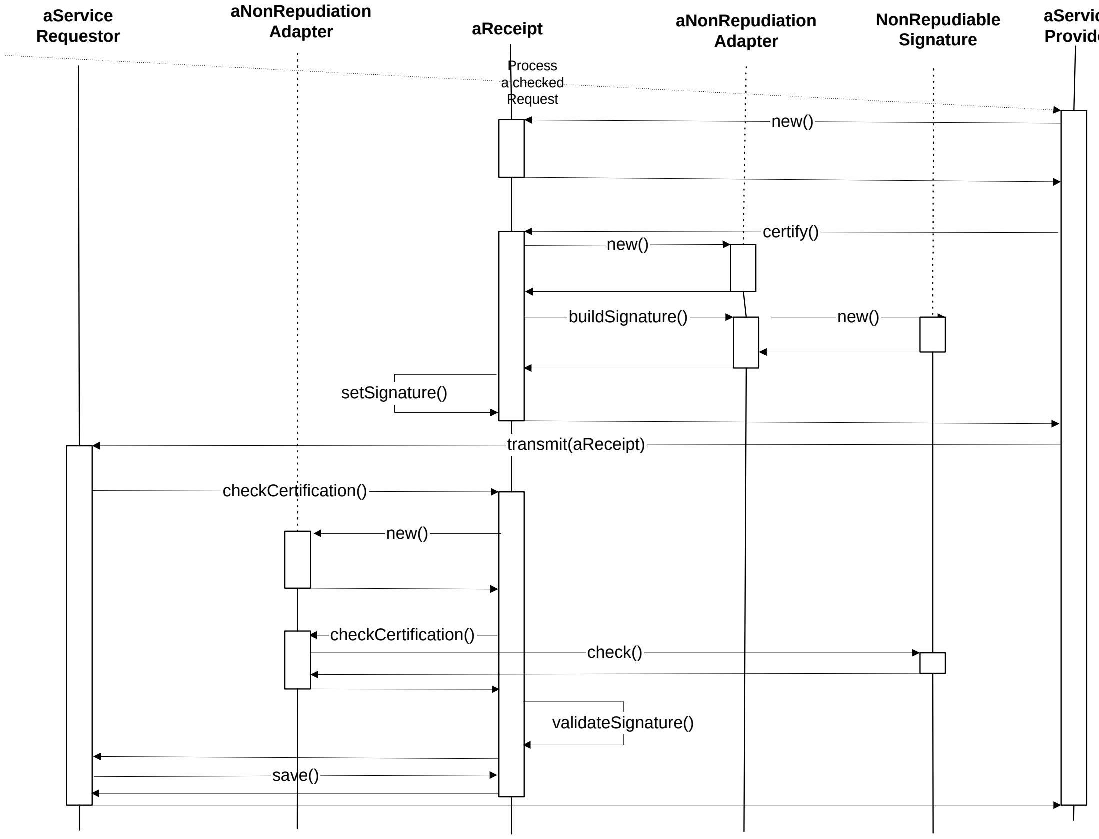
A ServiceRequestor has a Request to be processed by a ServiceProvider

1. The ServiceRequestor asks the Request to certify itself.
2. The Request creates a new NonRepudiaitonAdapter on the ServiceRequestor machine.
3. The Request asks for a NonRepudiableSignature for the NonRepudiationAdapter it has just created.
4. The NonRepudiationAdapter creates the NonRepudiableSignature.
5. The Request includes the NonRepudiableSignature to certify itself.
6. The ServiceRequestor sends the certified Request to the ServiceProvider.
7. The ServiceProvider asks the Request to check its own certification.
8. The Request creates a new NonRepudiaitonAdapter on the ServiceProvider machine.
9. The Request asks for a check of its NonRepudiableSignature to the NonRepudiationAdapter it has just created.
10. The NonRepudiationAdapter checks the NonRepudiableSignature.
11. The Request claims its own certification valid.
12. The ServiceProvider saves the Request.
13. The ServiceProvider processes the Request.

Scenario:

A ServiceProvider confirms to a ServiceRequestor that its Request has been processed in a non-repudiable fashion.

The following diagram shows the flow of calls needed to provide non-repudiation to a receipt.



Stimuli:

A ServiceProvider has processed a Request and must provide a Receipt to the Requestor.

1. The ServiceProvider creates a Receipt.
2. The ServiceProvider asks the Receipt to certify itself.
3. The Receipt creates a new NonRepudiaitonAdapter on the ServiceProvider machine.
4. The Receipt asks for a NonRepudiableSignature to the NonRepudiationAdapter it has just created.
5. The NonRepudiationAdapter creates the NonRepudiableSignature.
6. The Receipt includes the NonRepudiableSignature to certify itself.

7. The ServiceProvider sends the certified Receipt to the ServiceRequestor.
8. The ServiceRequestor asks the Receipt to check its own certification.
9. The Receipt creates a new NonRepudiaitonAdapter on the ServiceRequestor machine.
10. The Receipt asks for a check of its NonRepudiableSignature to the NonRepudiationAdapter it has just created.
11. The NonRepudiationAdapter checks the NonRepudiableSignature.
12. The Receipt claims its own certification valid.
13. The Receipt saves itself on the ServiceRequestor side.

Specification

Package: com.ac.eas.security.cryptography

Related Frameworks

Framework: Persistence