UNS Detailed Design

Prepared by

Nedeljko Pavlovic, European Dynamics

March 2008

Version 1.1

***Design document***

Version management

|  |  |  |  |
| --- | --- | --- | --- |
| No | Date | Changes | Author |
| 1.0 | 15/04/05 | Release to EEA | Nedeljko Pavlovic |
| 1.1 | 14/03/08 | Migrate to OpenDocument | Søren Roug |

*Project manager: Dr. Stelios Papastratos*

Contents

1. Introduction [6](#_toc1837)

1.1. Purpose [6](#_toc1838)

1.2. Scope [6](#_toc1840)

1.3. Overview [6](#_toc1842)

2. Decomposition description [7](#_toc1865)

2.1. Layer decomposition [7](#_toc1867)

2.1.1. Presentation layer [7](#_toc1870)

2.1.2. Business logic layer [8](#_toc1872)

2.1.3. Enterprise resources abstraction layer [8](#_toc1874)

2.1.4. Enterprise resources layer [8](#_toc1876)

2.2. Module decomposition [8](#_toc1879)

2.2.1. GUI module [9](#_toc1881)

2.2.2. Admin module [9](#_toc1898)

2.2.3. Subscription module [11](#_toc1937)

2.2.4. Harvester module [11](#_toc1943)

2.2.5. Notification module [12](#_toc1948)

2.2.6. Utility module [12](#_toc1954)

2.2.7. ERA module [12](#_toc1960)

2.2.8. Feed module [12](#_toc1962)

2.3. Concurrent process decomposition [12](#_toc1965)

2.3.1. Events harvester [13](#_toc1967)

2.3.2. Notifications daemon [13](#_toc1969)

2.4. Data decomposition [13](#_toc1971)

2.4.1. EEA User [13](#_toc1974)

2.4.2. EEA Role [14](#_toc1978)

2.4.3. Delivery type [14](#_toc1981)

2.4.4. Notification template [14](#_toc1984)

2.4.5. Notification profile type [14](#_toc1988)

2.4.6. Channel [14](#_toc1994)

2.4.7. Event [15](#_toc2004)

2.4.8. Subscription [15](#_toc2014)

2.4.9. Notification [15](#_toc2020)

2.4.10. Delivery [16](#_toc2025)

3. Dependency description [16](#_toc2031)

3.1. Intermodule dependencies [16](#_toc2032)

3.2. Data dependencies [17](#_toc2058)

3.2.1. Mapping Procedure [17](#_toc2059)

3.2.2. Additional remarks [18](#_toc2065)

4. Interface description [18](#_toc2070)

4.1. Module interface [18](#_toc2072)

4.1.1. GUI module [18](#_toc2073)

4.1.2. Admin Module [18](#_toc2075)

4.1.3. Subscription module [19](#_toc2086)

4.1.4. Harvester module [19](#_toc2088)

4.1.5. Notification module [19](#_toc2091)

4.1.6. Feed module [19](#_toc2093)

4.1.7. ERA module [19](#_toc2095)

5. Detailed design [20](#_toc2098)

5.1. Module detailed design [20](#_toc2099)

5.1.1. Admin module [20](#_toc2101)

5.1.2. Subscription module [27](#_toc2181)

5.1.3. Harvester module [28](#_toc2193)

5.1.4. Notification module [30](#_toc2226)

5.1.5. ERA module [32](#_toc2250)

5.1.6. Feed module [38](#_toc2345)

5.2. Data detailed design [39](#_toc2359)

5.2.1. Database schema [39](#_toc2360)

5.2.2. RDF schema [43](#_toc2382)

6. Deployment view [44](#_toc2436)

Appendix A - Interactions diagrams [47](#_toc2464)

Illustration Index

Illustration 1: Unified Notification Service modules and dependencies 8

Illustration 2: Admin module structure 10

Illustration 3: The UNS ER diagram 13

Illustration 4: The Channel management module class diagram 21

Illustration 5: The notifications profile typology management module class diagram 22

Illustration 6: The UNS Configuration management module class diagram 23

Illustration 7: The Data monitoring module class diagram 25

Illustration 8: The UI Templates module management module class diagram 26

Illustration 9: The Subscription module class diagram 27

Illustration 10: The Harvester module class diagram 28

Illustration 11: The Notification module class diagram 30

Illustration 12: The ERA module class diagram 32

Illustration 13: The Feed module class diagram 38

Illustration 14: The UNS Database physical data model diagram 39

Illustration 15: The UNS deployment diagram 45

Illustration 16: Create new channel sequence diagram 47

Illustration 17: Subscribe sequence diagram 48

Illustration 18: Pushing events to the UNS sequence diagram 49

Illustration 19: Harvesting events sequence diagram 49

Illustration 20: Sending notifications sequence diagram 50

Illustration 21: Start/Stop Harvesting sequence diagram 51

Illustration 22: Providing events feed to the external web services sequence diagram 51

# Document conventions

Acronyms/Abbreviations:

DTML - Document Template Mark-up Language

EEA - European Environment Agency

TSD - Technical Specifications Document

EIONET - The European Environmental Information and Observation

Network

HTTP - Hypertext Transfer Protocol

REQ - Each requirement is uniquely identified by a REQ and an

appropriate sequence number

RDF - Resource Description Framework

RSS - Really Simple Syndication

SMTP - Simple Mail Transfer Protocol

TCP/IP - Transmission Control Protocol / Internet Protocol

XML - Extensible Mark-up Language

XML-RPC - Remote procedure calling using HTTP as the transport protocol

and XML as the encoding protocol.

XMPP - Extensible Messaging and Presence Protocol

UI - User Interface

UNS - Unified Notification Service

ZODB - ZOPE Object Database

ZPT - Zope Page Templates

# Introduction

## Purpose

A Technical specifications document (TSD) is a representation of the software system that is used to facilitate analysis, planning, implementation, and decision making. This Technical specifications document (TSD) refers to the REPORTNET Unified Notification Service.

## Scope

The purpose of the notification service is to notify users on specific events, as well as on the approach of deadlines and dates. The notifications relate to the REPORTNET dataflow collection and handling cycle and users should be able to set up a notification profile in accordance to their personal preferences.

## Overview

The structure of the document is as follows:

Section 2 (Decomposition Description) offers several views on how to decompose the total structure of the REPORTNET Unified Notification Service System. The system can be decomposed in four different views:

* Layer decomposition;
* Module decomposition;
* Concurrent process decomposition;
* Data decomposition

In terms of layer decomposition the system is decomposed in the following layers:

* Presentation layer;
* Business logic layer;
* Enterprise resources abstraction layer;
* Enterprise resources layer.

In terms of module decomposition the system is decomposed in the following modules:

* GUI Module (belongs to presentation layer)
* Admin Module (belongs to business logic layer)
* Harvester Module (belongs to business logic layer)
* Notification Module (belongs to business logic layer)
* Utility Module (belongs to business logic layer)
* Subscription Module (belongs to business logic layer)
* ERA module (belongs to Enterprise resources abstraction layer)
* Feed Module(belongs to business logic layer)

Section 3 provides information about Intermodule and data dependencies. Section 4 provides a description of the interfaces for the various modules. Section 5 presents the detailed design of each module.

# Decomposition description

This section offers several views on how to decompose the total structure of the REPORTNET Unified Notification Service system. Emphasis will be placed on general layering, modularity, processes, and the general structure of data.

## Layer decomposition

The REPORTNET Unified Notification Service logical partitioning consists of application layers that collaborate together through well-defined interfaces, but do not share internal knowledge about each other. This not only makes the structure and design of the UNS easier to understand, but it also has scaling implications. Since each layer communicates with the one below it using well-known interfaces, it is a simple matter to replace any one of the layers as long as the new layer presents an identical interface.

The UNS consists of the layers described below:

### Presentation layer

The presentation layer handles the layout, formatting, and display of query results to users. This layer provides views of data to users of the UNS. The UNS presentation layer is concerned only with providing a user interface that enables the UNS users to interact with the system.

### Business logic layer

This layer implements most of the REPORTNET Unified Notification Service functions, which has the overall control of events and notifications processing and it provides a software representation of objects that users deal with when interacting with the UNS.

### Enterprise resources abstraction layer

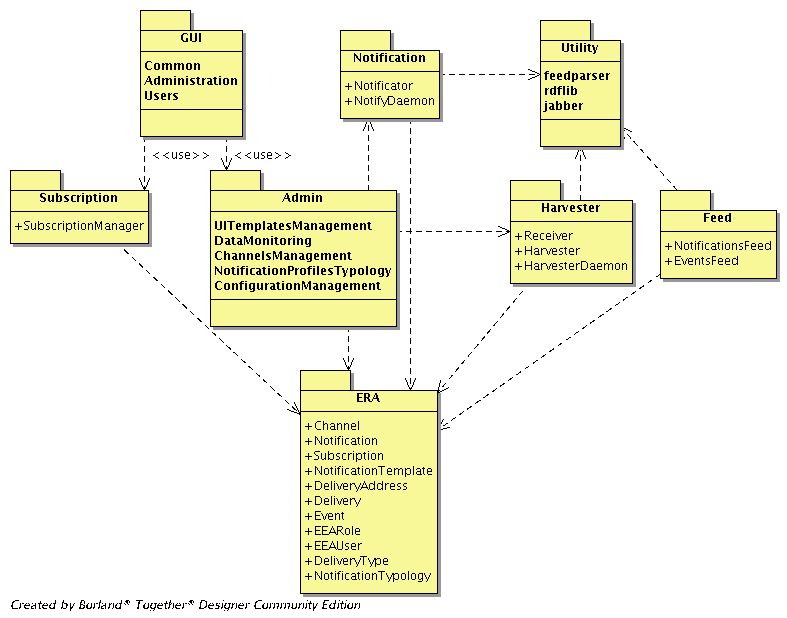
This is an object abstraction layer wrapping the enterprise resources. In the UNS design model, Python classes are normally used.

### Enterprise resources layer

This layer comprises non-Python resources storing raw data. It consists of MySQL Database, Zope Object Database, OpenLDAP and File system.

For each module described in the following 2.2.1 section the layer in which the module is placed will be pointed out.

## Module decomposition

Illustration 1: Unified Notification Service modules and dependencies

### GUI module

The GUI module represents the conceptual module that contains all objects related to the UNS user interface. Development of the UNS will enforce usage of ZPT as main objects that will participate in building this module. DTML may not be used at all.

Moreover, we will split this module in more conceptual modules as described in the sections bellow.

This module, as well as its leaf modules, belongs to the REPORTNET Unified Service presentation layer.

#### Administration GUI

The administration user interface will allow EIONET Administrators access to administrative functions of the UNS. The following list represents functions that will be accessible through the administrative user interface:

* Channels management
* Notification profiles management
* Data monitoring
* UNS configuration
* UI templates management

#### User’s GUI

User’s graphical user interface will allow EIONET Users to access the UNS functionalities responsible for subscription management.

#### Common GUI

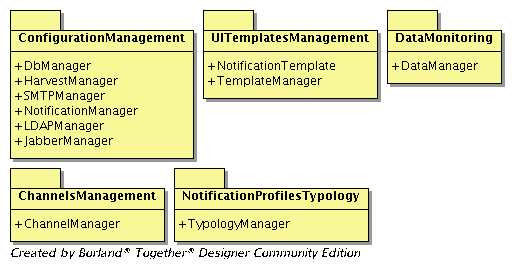
The common user interface will expose functionalities common to all users that may access the UNS web site. The following list represents these functions:

* Login
* Access to links related to other REPORTNET web services

### Admin module

The admin module is a conceptual module that contains core functionalities related to the EIONET Administrator. Physically it represents a Python package.

This module and its leaf modules belong to the REPORTNET Unified Service business logic layer.

Illustration 2: Admin module structure

#### Channel management module

This module provides functionalities related to events channels management. It will provide the following functions:

* Creation of new channels
* Editing of existing channels
* Removal of channels

This module belongs to the REPORTNET Unified Service business logic layer. Section 5.1.1.1 presents the detailed design of the channel management module.

#### Notification profiles typology management module

This module will provide functions to manage the typology of the notifications profiles. The following functions are provided by this module:

* Creation of new notification profile types
* Editing of existing notification profile types
* Removal of notifications profile types

This module belongs to the REPORTNET Unified Service business logic layer. Section 5.1.1.2 presents the detailed design of the notification profiles typology management module.

#### UNS configuration management module

The UNS configuration module will consist of the functions related to setting up the REPORTNET Unified Notification Service. In what follows we present the list of functions for this module:

* Set up Jabber server configuration parameters
* Set up SMTP configuration parameters
* Set up MySQL Database configuration parameters
* Set up LDAP Server configuration parameters
* Set up events harvester daemon
* Set up notifications daemon

This module belongs to the REPORTNET Unified Service business logic layer. Section 5.1.1.3 presents the detailed design of the UNS configuration management module.

#### Data monitoring module

The Reporting module will provide functions which will be used for reports generation, as well as for reacting on notification delivery failures.

The following list presents these functions:

* Notifications throughput report
* Failed notifications report
* Removal of problematic notifications profiles

This module belongs to the REPORTNET Unified Service business logic layer. Section 5.1.1.4 presents the detailed design of the Data monitoring module.

#### UI templates management module

This module will provide core functionalities related to the management of notification templates and the management of basic design elements of the UNS User interface. The functions provided by this module are the following:

* Create notification template
* Edit notification template
* Remove notification template

This module belongs to the REPORTNET Unified Service business logic layer. Section 5.1.1.5 presents the detailed design of the UI templates management module.

### Subscription module

This module will provide services related to the EIONET Users. Core functions it will provide are:

* Subscribing to the channels
* Edit existing subscriptions
* Remove subscriptions

This module belongs to the REPORTNET Unified Service business logic layer. Section 5.1.2 presents the detailed design of the Subscription module.

### Harvester module

The Harvester module contains core functionalities for harvesting EIONET events. It will provide the following functions:

* Automatic harvesting of events
* Push function for external web services

This module belongs to the REPORTNET Unified Service business logic layer. Section 5.1.3 presents the detailed design of the Harvester module.

### Notification module

The Notification module provides functions that are responsible for generating notifications and delivering these notifications. The following list represents the module functions:

* Automatic generation of notifications
* Delivering of notifications by using SMTP
* Delivering of notifications by using XMPP

This module belongs to the REPORTNET Unified Service business logic layer. Section 5.1.4 presents the detailed design of the Notification module.

### Utility module

The Utility module contains core functionalities provided by the following external libraries:

* RDF parser
* RSS parser
* Jabber API

The reason why it is deemed best to consider these libraries as part of the UNS source code is because they simplify the installation process of the UNS; in addition, they allow possible modifications in order to extend their functionalities. This module belongs to the REPORTNET Unified Service business logic layer.

### ERA module

This module contains classes providing object representation of the data entities the UNS deals with. All its classes provide functionalities for manipulation of their persistence. This module belongs to the REPORTNET Unified Service business enterprise resources abstraction layer. Section 5.1.5 presents the detailed design of the UNS ERA module.

### Feed module

This module contains classes responsible for providing events and notifications data in RDF form to the external web services.

This module belongs to the REPORTNET Unified Service business logic layer.

## Concurrent process decomposition

The Unified notification service will contain two distinct processes described in the sections bellow.

### Events harvester

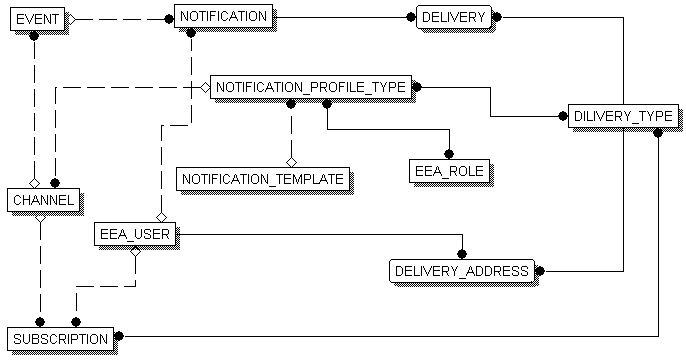
The first process of the UNS is the Event Harvester that gathers the event information for which notifications may be needed to be generated. This process will harvest events for the each channel and store these into the system database.

### Notifications daemon

The Notifications daemon will be the part of the UNS that will actually send out the notifications to the EIONET Users. At every invocation of the notification daemon a notification e-mail and/or a jabber message will be generated for each EIONET User depending on his/her preferences, available events and will be based on the notification templates as generated by the EIONET Administrators.

## Data decomposition

The data handled in the UNS consist of a few real world entities. These entities are described here conceptually. Therefore, this section contains a conceptual model of all data entities, which can then be readily translated into a database schema .The Database Schema will be presented in the section 5.2.1. Some of the entities will be translated to the RDF schema which will be described in the section 5.2.2. The following figure represents the UNS logical data model.

Illustration 3: The UNS ER diagram

### EEA User

An EEA user is someone who has been registered in EEA’s LDAP server.

* *ExternalUserId*: Unique identifier of the user used in the EEA LDAP server; currently, it is the username.
* *DeliveryAddresses:* The Deliver addresses representing actual addresses for each delivery type where notifications will be sent to the EEA user.

### EEA Role

EEA Users are grouped in EEA Roles in accordance to their privileges.

* ExternalRoleId: Unique identifier of the user used in EEA LDAP server; currently, it is the role name.

### Delivery type

Delivery type represents how notifications will be sent to the end users. The UNS will contain two delivery types Jabber and E-Mail. This entity will exist in order to allow future extensions to the UNS.

* *Name*: The name of the delivery type.

### Notification template

The Notification template represents the text used as a template for forming the notification messages.

* *Name*: The name of the notification template.
* *Content*: Text representing the template.

### Notification profile type

Notification profile type represents the entity used for classification of the event channels in accordance with user groups, delivery types and notification templates.

* *Name*: The name of the notification profile type
* *Template*: The one notification template related to profile type
* *AllowedRoles*: EEA roles allowed to subscribe to channels which belong to the profile type
* *AllowedDeliveryTypes:* The delivery types related to the notification profile type.

### Channel

A channel is a core concept of the UNS. A channel is an entity used for providing events data from an external web service to the UNS.

* *Name*: The name of the channel.
* *Description*: The description of the channel.
* *LastHarvest*: A timestamp representing the time the last harvest process occurred.
* *Mode*: A term representing the working mode of the channel. It is allowed to be ‘PUSH’ or ‘PULL’
* *FeedUrl*: The URL of the external feed; needed for channels that will work in pull mode.
* *NotificationProfileType*: The one notification profile type where the channel belongs.
* *RefreshDelay*: A number representing the number of seconds that need to expire before a new harvesting for the channel occurs. The default value will be 600 seconds; needed for channels that will work in pull mode.
* *Events*: Events collected and created for the channel.

### Event

The events are collected for each channel.

* *Channel*: Channel where the event belongs.
* *Title*: The title of the event
* *Description*: The description of the event.
* *ExternalId*: The external web service unique identifier of the event.
* StartDate: A timestamp representing when the event starts.
* EndDate: A timestamp representing when the event ends.
* *Causer:* The external web service which causes the event; for example it can be the activity name from the CDR workflow.
* *CreationDate:* A timestamp representing when the event has been inserted.

### Subscription

Each EIONET User may subscribe to the some of the existing channels related to the notification profile type the EIONET User roles belongs to. The subscription entity represents the subscription of a single EIONET User to a single channel.

* *EEAUser*: The EEA user that made subscription.
* *Channel:* The event content channel the user subscribed to.
* *LeadTime:* A number representing seconds for defining the lead time for notification forewarn.
* *DeliveryTypes:* The Deliver types representing how notifications will be sent to the EIONET User.

### Notification

Notifications are the messages generated for subscribed EIONET Users and ready for delivering.

* *Event*: The event for which a notification is generated.
* *EEAUser*: The EEA user subscribed to receive a notification
* *Body*: Text representing the notification message.

### Delivery

Delivery represents an attempted delivery of the notification by using a specified delivery type.

* *Notification:* Notification to be sent by delivery
* *DeliveryType:* Represents how a notification is going to be delivered to the EEA User.
* *DeliveryStatus:* A number representing the status of delivery. It can be 1 for successful delivery and 0 for unsuccessful delivery.
* *DeliveryTime*: A timestamp representing the time when the delivery process occurred.

# Dependency description

## Intermodule dependencies

The REPORTNET Unified Service modules depend on the following list of the external libraries and Zope products:

Python libraries:

* ittools version 0.7.3
* PyXML version 0.8.4
* Python-ldap version 2.0.6
* MySQL-python version 1.2.0
* Rdflib version 2.0.6
* Jabberpy version 0.5
* Feedparser version 3.3

Rdflib, Jabberpy and feedparser will be imported in the UNS Zope product source structure under the utility module.

Zope products:

* iHotfix version 0.5.2
* Localizer version 1.1.0
* LDAPUserFolder version 2.5[[1]](#footnote-2)

The following modules rely on other modules that are part of the REPORTNET Unified Notification Service:

* GUI Module. This module depends on the Admin and Subscription module. Conceptually this dependence represents the connection between the UNS presentation and the business logic layer.
* *Administration GUI module.* This module uses the Admin module in order to accomplish tasks requested by the EIONET Administrators
* *Users GUI module.* This module uses the Subscription module in order to accomplish tasks requested by the EIONET Users.
* *Common GUI module.* This module uses the LDAPUserFolder product in order to allow users to login into the UNS.
* *Harvester module.* This module uses the Utility module in order to process RSS/RDF streams pulled/pushed form external web services. It also depends on the ERA module in order to work with object representation of the data entities and manage their persistence.
* *Notification module.* This module uses the Utility module in order to process jabber messages. It also depends on the ERA module in order to work with object representation of the data entities and manage their persistence.
* *Subscription module.* This module depends on the ERA module.
* *Admin module.* This module depends on the ERA, Harvester and Notification modules.
* *Feed module.* This module depends on the ERA and Utility modules.

## Data dependencies

### Mapping Procedure

The mapping of the conceptual domain model to the database schema will be constructed using the following rules:

1. Every concept should correspond with at least one entity.
2. Attributes are mapped directly if they represent a type of attributes that is native to the MySQL 4.1 Database System.
3. Attributes that refer to concepts in the domain model are mapped to a foreign key attribute that uniquely identifies the entity in the schema.
4. If the relationship between entities has a many-to-many role this relationship is expanded into a separate table, where both foreign keys are united as the primary key of the relation.

### Additional remarks

The REPORTNET Unified notification service is going to support EEA Roles and EEA Users kept on the EEA LDAP server. Notification profiles typology depends on EEA Roles and Subscriptions/Notifications depends on EEA Users. Since these entities are going to be kept in the UNS Database it seems that no problem will be encountered. However, a question raised here is how the lifecycle of the EEA Roles and the EEA Users on the EEA LDAP server will reflect on the UNS. The UNS system design tackles this question by the following approaches:

1. If a user wants to subscribe to some of the channels allowed to his/her EEA Role the UNS will check for the existence of the EEA User in the UNS Database. In case that the user does not exist it will be created. In each invocation of the Notification daemon process, the list of EEA LDAP users will be cached and synchronized with the UNS database.
2. Since the Subscription depends on the Notifications profiles typology, the best place for handling the EEA Roles is in the management of the notifications profile typology. When an EIONET Administrator access the notification profiles typology management section the UNS will contact the EEA LDAP server and perform a synchronization of the EEA Roles between the UNS Database and the LDAP server.

# Interface description

This subsection describes everything designers, programmers and testers need to know in order to use correctly the functions provided by the UNS modules. It includes the details of internal interfaces not provided in the REPORTNET Unified Service Functional Specifications Document.

## Module interface

### GUI module

The interface to the REPORTNET Unified Notification Service GUI module and all its leaf modules is provided by the Zope application server. The GUI module will be accessed by the EEA users.

### Admin Module

#### Channel management module

The interface to the Channel management module is provided by the public methods of the ChannelManager class. The Administration GUI module uses this interface.

#### Notification profiles typology management module

The interface to the Notification profiles typology management module is provided by the public methods of the TypologyManager class. The Administration GUI module uses these interfaces.

#### UNS configuration management module

The interface to the UNS configuration module is provided by the public methods of the NotificationManager, HarvestManager, LDAPManager, SMTPManager and DbManager classes the Administration GUI module uses these interfaces.

#### Data monitoring module

The interface to the Channel management module is provided by the public methods of the DataManager class. The Administration GUI module uses this interface.

#### UI templates management module

The interface to the UI templates management module is provided by the public methods of the TemplateManager class. The Administration GUI module uses this interface.

### Subscription module

The interface to the Subscription management module is provided by the public methods of the SubscriptionManager class. The Users GUI module uses this interface.

### Harvester module

The interface to the Subscription management module is provided by the public methods of the Harvester class. The UNS configuration module and possibly the Linux system crone job scheduler use this interface.

The harvester module will have one more interfaces that will be used by external web services that will push events data to the UNS. This interface is provided by the public method of the Receiver class. This method will be called by the XML-RPC protocol.

### Notification module

The interface to the Subscription management module is provided by the public methods of the Notificator class. The UNS configuration module and possibly the Linux system crone job scheduler use this interface.

### Feed module

The interface to the Feed module is provided by the callable classes EventsFeed class and NotificationsFeed class. The external EIONET Web services use this interface in order to retrieve information about events and notifications stored in the UNS database.

### ERA module

The interface to the ERA module is provided by the public methods of all classes representing data entities contained in this module. Section 5.1.5 contains detailed description of these methods.

Module interfaces will be used to allow interaction between system modules and interactions between external systems and the UNS. These interactions are presented on interaction diagrams in Appendix A - Interactions diagrams.

# Detailed design

## Module detailed design

The section presents the internal details of each UNS module. These details include attribute descriptions for identification, processing and data. It also contains the details that will be needed by the programmers for the relevant implementations.

### Admin module

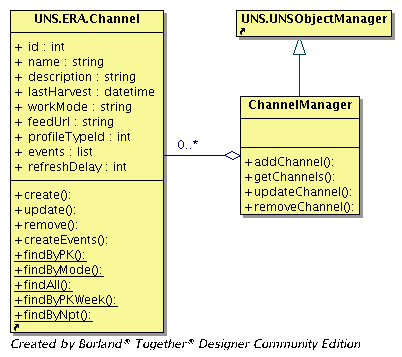
The UNSObjectManager class is represented in all class diagrams regarding the admin module. UNSObjectManager class is mix-in class of the UNS product, which subclasses the following Zope engine basic classes: Acquisition.Implicit, Persistent and Traversable. Each class specific for the Admin modules inherits the UNSObjectManager class.

Characteristics of these parent classes are:

* Class Persistence provides persistence of object state even after restart of Zope server or after un unexpected crash occurs;
* Class Acquisition.Implicit provides implicit acquisition for its subclasses;
* Class Traversable provides common methods related to physical and virtual path of published objects.

UNSObjectManager class is oriented to manipulate with the UNS product administrative tasks such as creation and identification of other subclasses and their paths in the UNS product.

#### Channel management module

Illustration 4: The Channel management module class diagram

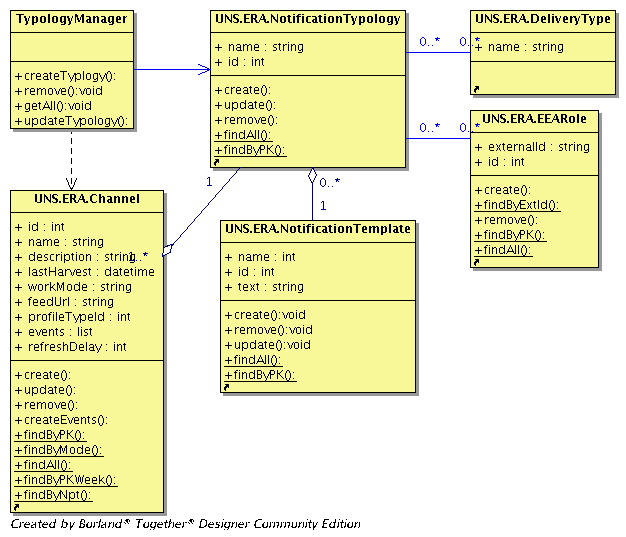
##### ChannelManager class

**Attributes**: No attributes

Methods:

* *addChannel(self,REQUEST)*  
  This method takes and validates parameters from the REQUEST object concerning a new channel and creates the new Channel object. Then it invokes the create method of the Channel in order to create a new channel in the UNS database. The REQUEST object contains the complete information about the current request made to the Zope.
* *getChannels(self)*Calls the *findAll* static method of the Channel class,retrieves and returns to the caller the list of theChannelobjects representing the UNS existing channels.
* *updateChannel(self,REQUEST)*  
  This method takes and validates parameters representing existing channel details from the REQUEST object and creates the new Channel object. It then invokes the update method of the Channel object in order to store new information inside the UNS database.
* removeChannel*(self,REQUEST)*This method takes and validates the parameter representing the Channel unique identifier, creates a new Channel object and delete a channel from the UNS database.

#### Notifications profiles typology management module

Illustration 5: The notifications profile typology management module class diagram

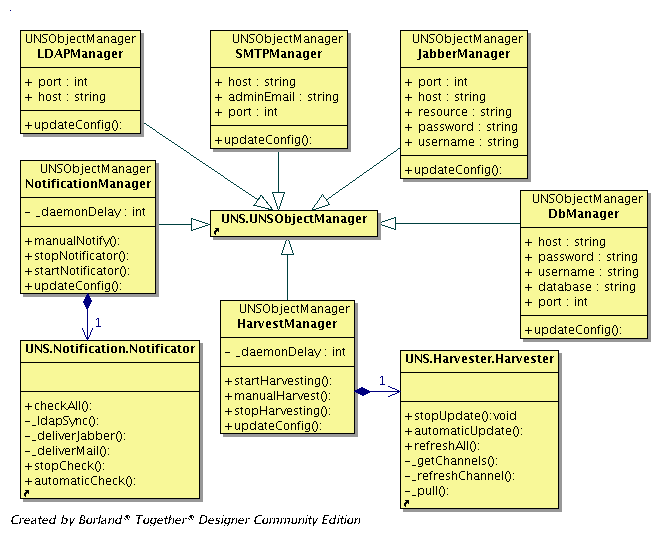
##### TypologyManager class

**Attributes**: No attributes

Methods:

* *addTypology(self,REQUEST)*  
  This method takes and validates parameters from the REQUEST object about a new notification profile type and creates the new NotificationTypology object. It then invokes the create method of the Channel in order to create a new channel in the UNS database.
* *removeTypology(self,REQUEST)*Calling the findAll static method of the Channel classretrieves and returns to the caller the list of theChannelobjects representing the UNS existing channels.
* *updateTypologyl(self,REQUEST)*  
  This method takes and validates the parameters representing existing channel details from the REQUEST object and creates the new Channel object. It then invokes the update method of the Channel object in order to store new information inside the UNS database.
* *getTyplogies(self)*This method takes and validates the parameter representing the Channel unique identifier, creates a new Channel object and invokes its remove method in order to delete a channel from the UNS database.

#### UNS Configuration management module

Illustration 6: The UNS Configuration management module class diagram

##### HarvestManager class

Manages the channels’ harvesting daemon. Persistence of this class object is automatic and related to the ZODB (ZOPE Object Database).

**Attributes**:

- \_daemonDelay  
Harvester daemon wake up interval.

Methods:

* *updateConfig(self,REQUEST)*  
  This method takes and validates the parameter from the REQUEST object representing the harvester daemon wake up interval and sets the daemonDelay attribute.
* *startHarvesting(self)*Creates Harvester class object and start up Harvester daemon.
* *stopHarvesting(self)  
  S*tops the Harvester daemon.
* *manuallNotify*(self)  
  It first checks if the Harvester daemon process is active; if not it invokes the events harvesting otherwise it returns an error message concerning the already active process.

##### NotificationManager class

Manages the notification of the daemon. Persistence of this class object is automatic and related to the ZODB.

**Attributes**: No attributes

Methods:

* *updateConfig(self,REQUEST)*  
  This method takes and validates the parameter from the REQUEST object representing the notification daemon wake up interval and sets the daemonDelay attribute.
* *startNotificator(self)*CreatesNotificator class object and starts up the Notification daemon.
* *stopNotificator(self)  
  S*tops the Notification daemon.
* *manuallNotify*(self)  
  It first checks if the Notification daemon process is active; if not it invokes the notifications generation otherwise it returns an error message concerning the already active process.

##### SMTPManager class

Manages the configuration parameters related to the SMTP server that will be used for notification delivery by using e-mail messages. Persistence of this class object is automatic and related to the ZODB.

##### JabberManager class

Manages the configuration parameters related to the Jabber server that will be used for notification delivery by using Jabber messages. Persistence of this class object is automatic and related to the ZODB.

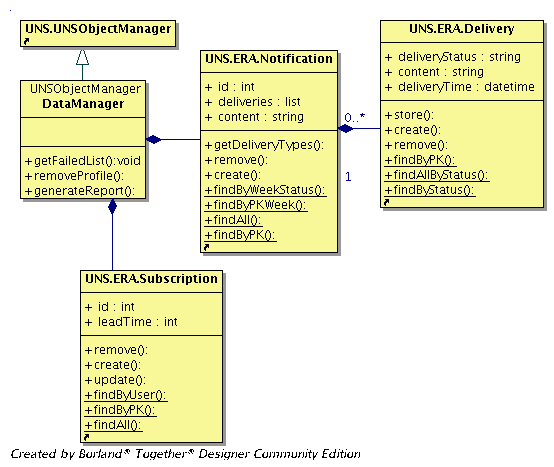
##### DbManager class

Manages the configuration parameters related to the MySQL database server that will be used as persistence storage by the UNS. Persistence of this class object is automatic and related to the ZODB.

##### LDAPManager class

Manages configuration parameters related to the LDAP server that will be used for the authentication purposes, as well as for retrieval of EEA Roles and EEA Users. Persistence of this class object is automatic and related to the ZODB.

#### Data monitoring module

Illustration 7: The Data monitoring module class diagram

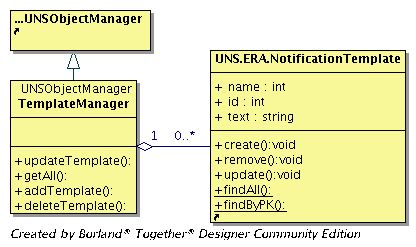
##### DataManager class

**Attributes**: No attributes

Methods:

* *generateReport(self,REQUEST)*  
  Takes and validates the parameter from the REQUEST object representing the parameters for the Notifications throughput report generation. Then it creates the list of the Notification class objects and returns it to the caller. Parameters that may be provided are date interval, delivery type, delivery status. The method must examine the persistence of these parameters and generate the appropriate query that will return the appropriate notifications.
* *getFailedList (self)*Returns the list of the Notification objects that have unsuccessful delivery attempts.
* *removeProfilel(self,REQUEST)*  
  Removes the subscription related to the provided notification. Takes from the REQUEST object the unique identifier of the problematic notification.

#### UI Templates management module

Illustration 8: The UI Templates module management module class diagram

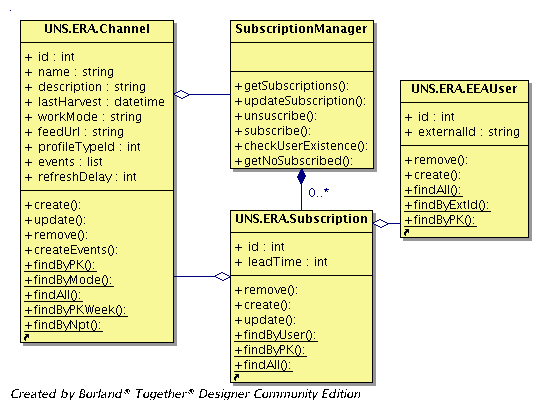
##### TemplateManager class

**Attributes**: No attributes

Methods:

* *getAll(self)*  
  Returns the list of the NotificationTemplate objects. The list is formed by calling the static method *findAll* of the NotificationTemplate class.
* *addTemplate(self,REQUEST)*Takes parameters from the REQUEST object representing the name and the content of the new notification template. It then creates a new NotificationTemplate object and stores its attributes in the UNS database by calling it’s *create* method.
* *removeProfilel(self,REQUEST)*  
  Takes a parameter from the REQUEST object representing the unique identifier of the existing notification template and removes it from the UNS database by calling the *remove* method of the NotificationTemplate class object.
* *updateTemplate(self,REQUEST)*Takes parameters from the REQUEST object representing the unique identifier, name and content of the notification template. It then updates a new NotificationTemplate object and stores changes in the UNS database by calling its update method.

### Subscription module

Illustration 9: The Subscription module class diagram

#### SubscriptionManager class

**Attributes**: No attributes

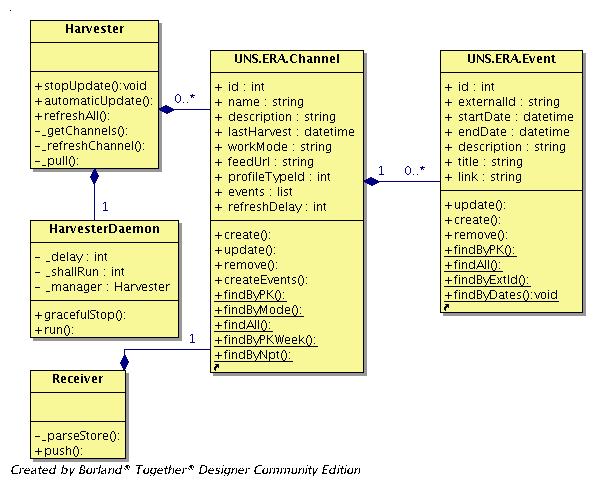
Methods:

* *getSubscriptions(self)*  
  Takes the logged in username, generates and returns the list of the Channel class objects representing existing subscriptions of the logged EIONET user.
* *getNoSubscribed (self)*

Takes the logged in username, generates and returns the list of the Channel class objects representing the channels assigned to the notification profile typology where the logged EIONET user role belongs.

* *subscribe(self,REQUEST)*This method takes and validates a parameter from the REQUEST object representing the unique identifier of the channel, username of the logged EIONET user, lead time, desired delivery types and delivery types addresses. It then creates a new subscription for the logged EIONET User.
* *subscribe(self,REQUEST)*  
  Removes a subscription for the logged EIONET User.

### Harvester module

Illustration 10: The Harvester module class diagram

#### Harvester class

**Attributes**: No attributes

Methods:

* *automaticUpdate(self,delay=10)*  
  Creates an instance of the HarvesterDaemon class and starts a thread.
* *stopUpdate(self)*Stops periodic refreshing of the channels by stopping the HarvesterDaemon thread.
* *refeshAll(self)*  
  Gets all existing UNS channels working in PULL mode and invokes the Harvester class method *refreshChannel* for each one.
* \_\_refreshChannel(self,channel)  
  For a given channel as argument, this method checks if the channel *refreshDelay* attribute value has expired. If yes it calls the pull method otherwise it finishes work.
* \_\_pull(self,channel)  
  For a given channel as argument this method checks pulling the external web service channel content, parses it, creates a list of events objects and stores their data in the UNS database.
* \_\_getChannels()  
  Gets all channels working in pull mode by calling the Channel class static function *findByMode*

#### HarvesterDaemon class

This class represents the thread that is responsible for periodical scanning of the channels.

**Attributes**:

* \_delay  
  a number representing the number of seconds a thread will sleep between innovations.
* \_manager  
  Holds an instance of the Harvester class.
* shallRun  
  Default value 1 otherwise can be set up to 0. Represents running indicator for the thread.

Methods:

* *run()*  
  The run method gives a thread to call the *refreshAll* method of the Harvester object. After method execution it will sleep for the time defined by the \_delay variable.
* *gracefulStop()*  
  The method instruct HarvesterDaemon thread to stop when wake up next time.

#### Receiver class

External EIONET web services will be able to push events for the existing channels by calling the *push* method of this class through the XML-RPC protocol.

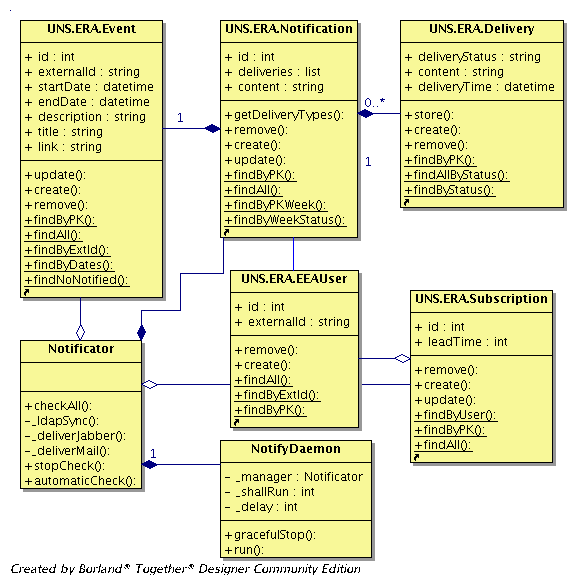
**Attributes**:

* No attributes

Methods:

* *push(self,rdf,channelId)*This method will be called through XML-RPC from external web services in order to push events data for a specified channel. The rdf parameter represents rdf formed content of the message and *channelId* represents the UNS existing channelId that is configured to work in PUSH mode. The method takes an instance of the Channel class by calling the Channel class static method *findByPK* and calls the *parseStore* method.
* *parseStore(self,rdf,channel)*Parses the RDF content provided, forms the Event class objects and stores these in the UNS database.

### Notification module

Illustration 11: The Notification module class diagram

#### Notificator class

**Attributes**: No attributes

Methods:

* *automaticCheck(self,delay=10)*  
  Creates an instance of the NotifyDaemon daemon class and starts a thread.
* *stopCheck(self)*Stops periodic refreshing of the channels by stopping the HarvesterDaemon thread.
* *checkAll(self)*Performs synchronization between EEA LDAP and the UNS database by calling the ldapSync method. It then retrieves all existing subscriptions and all events for which notifications have not been generated before and generates a list of the Notification objects in accordance with the user subscription preferences. It then assigns to the list of notifications objects all the notifications which have not been successfully delivered but have been created in previous invocation of the thread. At the end it iterates through the formed list and at each iteration stores the notification information in the UNS database - in case of new notifications - and tries to deliver the notification in accordance to the user subscription preferences.
* *\_ldapSync (self)*This method caches all EEA Users stored in the EEA LDAP server and performs synchronization with the UNS database*.*
* *\_deliverMail (self,notification)* Delivers notification by using SMTP and stores delivery information in the UNS database. Method delivers mail directly to the destination SMTP server. In case of unsuccessful delivery it will get error form the destination SMTP server.
* *\_deliverJabber (self,notification)* Delivers notification by using XMPP and stores delivery information in the UNS database.

#### NotifyDaemon class

Represents the thread responsible for invoking the periodical scanning of the existing subscriptions for which notifications shall be generated and delivered.

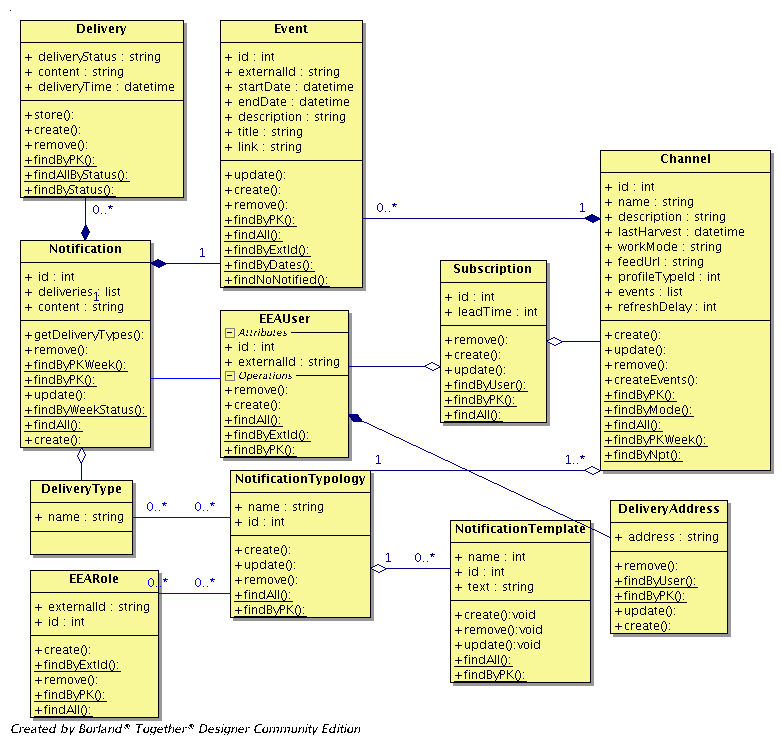
**Attributes**:

* \_delay  
  a number representing the number of seconds the thread will sleep between invocations.
* \_manager  
  holds an instance of the Notificator class.
* shallRun  
  Default value 1 otherwise can be set up to 0. Represents running indicator for thread

Methods:

* *run()*  
  The run method gives a thread to call the *automaticCheck* method of the Notificator object. After method execution it will sleep for the time defined by the \_delay variable.
* *gracefulStop()*  
  The method instruct NotifyDaemon thread to stop when wake up next time.

### ERA module

Illustration 12: The ERA module class diagram

Since the ERA module represents the object mapping of the data entities described in section 2.4 we will only specify the methods of the ERA classes.

All methods of the ERA class have the conn parameter representing the database connection provided from the business logic layer class which made the call. This upper layer classes are supposed to control database connections, as well as transactions.

#### Channel class

**Attributes**:

Methods:

* *create(self,conn)*  
  Creates the new Channel in the UNS database. It will take attributes values set up before its invocation.
* *remove(self,conn)*  
  Deletes the existing Channel from the UNS database.
* *update(self,conn)*  
  Updates the attributes of the existing Channel in the UNS database.
* *findByPK(conn,id)*  
  A static method that returns a Channel object representing the existing channel from the UNS database. The Id parameter represents the unique identifier of the existing channel.
* *findAll(conn)*  
  A static method that returns a list of the Channel objects representing all the existing channels stored inside the UNS database.
* *findByMode(conn,cmode)*  
  A static method that returns a list of the Channel objects representing the existing UNS database channels which are working in a mode specified by the cmode parameter.
* *findByPKWeek(conn,id)*  
  A static method that returns a Channel object representing the existing channel from the UNS database together with all the events gathered for this channel. The Id parameter represents the unique identifier of the existing channel.
* *findByNpt(conn,user)*  
  Static method that returns list of channel objects object representing channels to which logged user may subscribe but he/her did not.

#### Event class

**Attributes**:

Methods:

* *create(self,conn)*  
  Creates the new Event in the UNS database. It will take attributes values set up before its invocation.
* *remove(self,conn)*  
  Deletes the existing Event from the UNS database.
* *update(self,conn)*  
  Updates the attributes of the existing Event in the UNS database.
* *findByPK(conn,id)*  
  A static method that returns an Event object representing an existing event in the UNS database. The Id parameter represents the unique identifier of the existing event.
* *findAll(conn)*  
  A static method that returns a list of the Event objects representing all the existing events stored inside the UNS database.
* *findByExtId(conn,externalId)*  
  A static method that returns an Event object representing the existing event in the UNS database. The ExternalId parameter represents the external event provider unique identifier of the event.
* *findByDates(conn,midDate,maxDate)*  
  A static method that returns a list of the Events objects representing the existing events from the UNS database which are created between mindate and maxdate parameters.
* *findNoNotified(conn,channelId,userId)*  
  A static method that returns list of the Event object representing the events produced by channel with the id(*channelId*) that are not sent as notifications to the user specified with the *userId parameter.*
* *findForFeed(conn,\*args)*  
  A static method that returns list of the Event object representing the events queried from the UNS database by using arguments provided with the parameter *args*. This method will assume that id of the channel and the events creation dates interval may be present.

#### EEAUser class

**Attributes**:

Methods:

* *create(self,conn)*  
  Creates the new EEAUser in the UNS database. It will take attributes values set up before its invocation.
* *remove(self,conn)*  
  Deletes the existing EEAUser from the UNS database.
* *findByPK(conn,id)*  
  A static method that returns a EEAUser object representing an existing EEA user in the UNS database. The Id parameter represents the unique identifier of the existing user.
* *findAll(conn)*  
  A static method that returns a list of the EEAUser objects representing all the existing users stored inside the UNS database.
* *findByExtId(conn,externalId)*  
  A static method that returns an EEAUser object representing an existing user in the UNS database. The ExternalId parameter represents the external LDAP unique identifier of the user.

#### EEARole class

**Attributes**:

Methods:

* *create(self,conn)*  
  Creates the new EEARole in the UNS database. It will take attributes values set up before its invocation.
* *remove(self,conn)*  
  Deletes the existing EEARole from the UNS database.
* *findByPK(conn,id)*  
  A static method that returns an EEARole object representing an existing EEA role in the UNS database. The Id parameter represents the unique identifier of the existing role.
* *findAll(conn)*  
  A static method that returns a list of the EEARole objects representing all the existing roles stored inside the UNS database.
* *findByExtId(conn,externalId)*  
  A static method that returns an EEARole object representing an existing role in the UNS database. The ExternalId parameter represents the external LDAP unique identifier of the role.

#### NotificationTemplate class

**Attributes**:

Methods:

* *create(self,conn)*  
  Creates the new NotificationTemplate in the UNS database. It will take attributes values set up before its invocation.
* *remove(self,conn)*  
  Deletes the existing NotificationTemplate from the UNS database.
* *update(self,conn)*  
  Updates the attributes of the existing NotificationTemplate in the UNS database.
* *findByPK(conn,id)*  
  A static method that returns a NotificationTemplate object representing an existing notification template in the UNS database. The Id parameter represents the unique identifier of the existing template.
* *findAll(conn)*  
  A static method that returns a list of the NotificationTemplate objects representing all the existing notification templates stored inside the UNS database.

#### Notification class

**Attributes**:

Methods:

* *create(self,conn)*  
  Creates the new Notification in the UNS database. It will take attributes values set up before its invocation.
* *remove(self,conn)*  
  Deletes the existing Notification from the UNS database.
* *update(self,conn)*  
  Updates the attributes of the existing Notification in the UNS database.
* *findByPK(conn,id)*  
  Static method that returns Notification object representing existing notification in the UNS database. Id parameter represents unique identifier of the existing notification.
* *findAll(conn)*  
  Static method that returns list of the Notification objects representing all the existing notifications stored inside the UNS database.
* *findByPKWeek(conn,id)*  
  Static method that returns Notification object representing existing notification from the UNS database together with all deliveries for this notification. Id parameter represents unique identifier of the existing notification.
* *findByWeekStatus(conn,status)*Static method that returns Notification object representing existing notification from the UNS database together with all deliveries for this notification that have specified status.

#### Delivery class

**Attributes**:

Methods:

* *create(self,conn)*  
  Creates the new Delivery in the UNS database. It will take attributes values set up before its invocation.
* *remove(self,conn)*  
  Deletes the existing Delivery from the UNS database.
* *update(self,conn)*  
  Updates the attributes of the existing Delivery in the UNS database.
* *findByPK(conn,id)*  
  Static method that returns Delivery object representing existing delivery in the UNS database. Id parameter represents unique identifier of the existing delivery.
* *findByStatus(conn,status,notificationId)*Static method that returns list of the Delivery object representing existing deliveries with provided status and the notificationid from the UNS database.
* *findAllByStatus(conn,status)*Static method that returns list of the Delivery object representing existing deliveries with the provided status from the UNS database.

#### NotificationTyplogy class

**Attributes**:

Methods:

* *create(self,conn)*  
  Creates the new NotificationTypology in the UNS database. It will take attributes values set up before its invocation.
* *remove(self,conn)*  
  Deletes the existing NotificationTypology from the UNS database.
* *update(self,conn)*  
  Updates the attributes of the existing NotificationTypology in the UNS database.
* *findByPK(conn,id)*  
  Static method that returns NotificationTypology object representing existing notification profile type in the UNS database. Id parameter represents unique identifier of the existing notification profile type.
* *findAll(conn)*  
  Static method that returns list of the NotificationTypology objects representing all the existing notification profile types stored inside the UNS database.

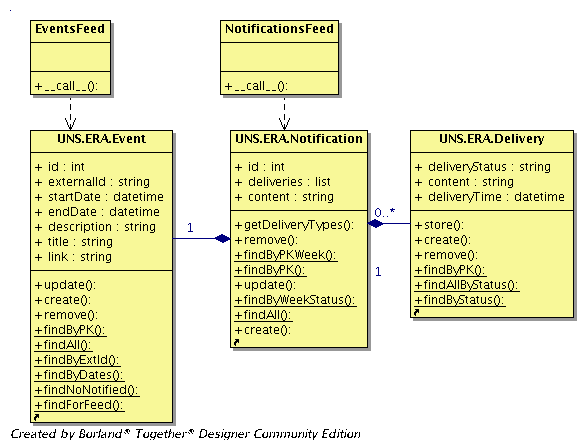
#### Subscription class

**Attributes**:

Methods:

* *create(self,conn)*  
  Creates the new Subscription in the UNS database. It will take attributes values set up before its invocation.
* *remove(self,conn)*  
  Deletes the existing Subscription from the UNS database.
* *update(self,conn)*  
  Updates the attributes of the existing Subscription in the UNS database.
* *findByPK(conn,id)*  
  Static method that returns Subscription object representing existing subscription in the UNS database. Id parameter represents unique identifier of the existing subscription.
* *findAll(conn)*  
  Static method that returns list of the Subscription objects representing all the existing subscriptions stored inside the UNS database.
* *findByUser(conn,extUserId)*  
  Static method that returns Subscription object representing existing subscription in the UNS database for the specified user. ExtUserId represents username of the EIONET User.

### Feed module

Illustration 13: The Feed module class diagram

#### EventsFeed class

Callable class that provides the external web services with the capability to access the UNS events data by using XML-RPC.

**Attributes**: No attributes

Methods:

* *\_\_call\_\_(self,REQUEST)*  
  Returns text representing RDF which contains the UNS events data.  
  It takes and validates the following parameters from the REQUEST object: channelId, minCreationDate, maxCreationDate. All these parameters will be optional but in cases that these will be provided, the method will use them to query the UNS database in order to retrieve events data.

#### NotificationsFeed class

Callable class that provides the external web services with the capability to access the UNS notifications data by using XML-RPC

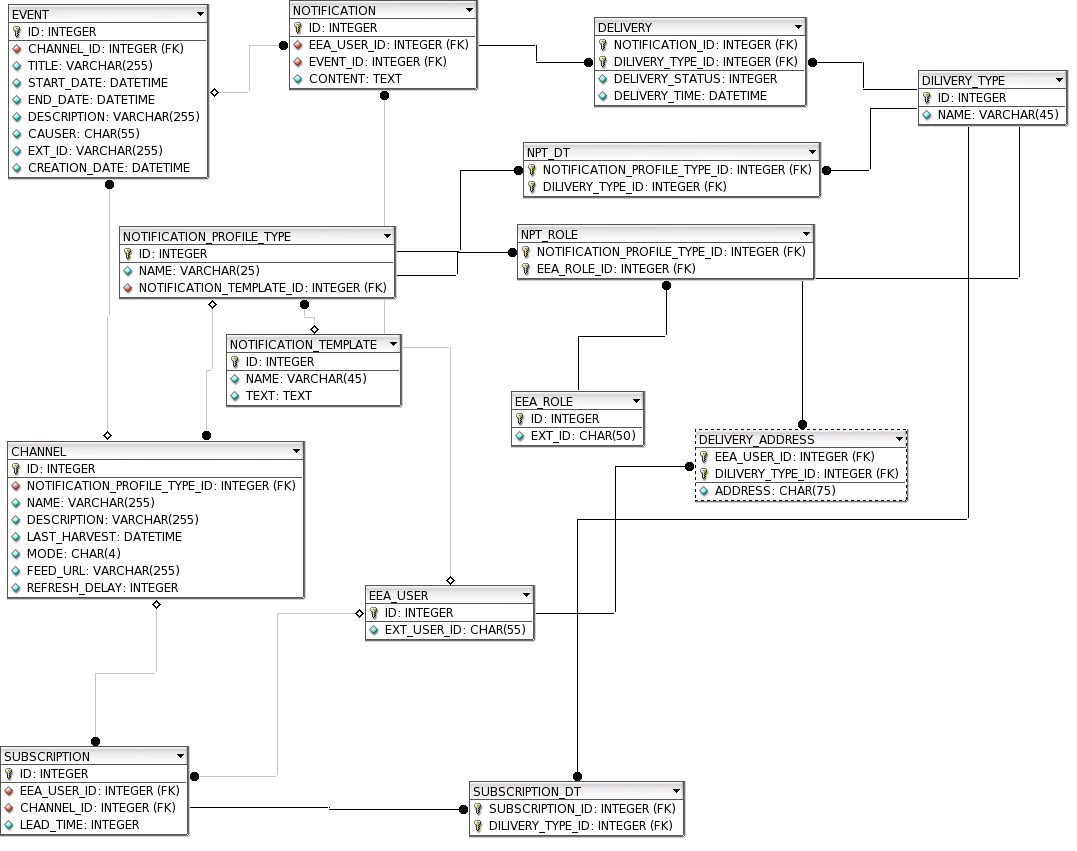
**Attributes**: No attributes

Methods:

* *\_\_call\_\_(self,REQUEST)*  
  Returns text representing the RDF which contains the UNS notifications data .  
  It takes the user parameter from the REQUEST objects and returns notifications generated for this user. The user parameter represents the username of the EEA user. Since the UNS will contain a large number of notifications the user will be a required parameter in order to reduce the amount of data returned to the remote caller.

## Data detailed design

### Database schema

Illustration 14: The UNS Database physical data model diagram

This section aims at providing a detailed description of the physical data model of the REPORTNET Unified Notification Service.

All the UNS Database tables are described in detail in the tables bellow.

# CHANNEL

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the subscription | AI |
| NOTIFICATION\_PROFILE\_TYPE\_ID | INTEGER |  | NN | UNSIGNED |  | Unique identifier of the referenced notification profile type |  |
| NAME | VARCHAR(255) |  | NN |  |  | The name of the channel |  |
| DESCRIPTION | VARCHAR(255) |  |  |  |  | A description of the channel |  |
| LAST\_HARVEST | DATETIME |  |  |  |  | A datetime representing the time when last harvest process occurred |  |
| MODE | CHAR(4) |  | NN |  | PULL | A term representing working mode of channel. It is allowed to be 'PUSH' or 'PULL' |  |
| FEED\_URL | CHAR(255) |  |  |  |  | The URL of the external feed needed for channels that will work in pull mode. |  |
| REFRESH\_DELAY | INTEGER |  |  | UNSIGNED | 600 | A number representing the number of seconds that need to expire before a new harvesting for the channel will occur |  |

# DELIVERY

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| NOTIFICATION\_ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the referenced notification |  |
| DILIVERY\_TYPE\_ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the referenced delivery type |  |
| DELIVERY\_STATUS | INTEGER |  | NN | UNSIGNED |  | A number representing the delivery status. It shall take value 1 for successful deliver and 0 for unsuccessful delivery. |  |
| DELIVERY\_TIME | DATETIME |  | NN |  |  | A datetime representing the time when the delivery process occurred |  |

# DELIVERY\_ADDRESS

DELIVERY\_TYPE

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| EEA\_USER\_ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the referenced EEA User |  |
| DILIVERY\_TYPE\_ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the referenced delivery type |  |
| ADDRESS | CHAR(75) |  |  |  |  | Address for single delivery type where notifications shall go |  |

# EEA\_ROLE

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the notification | AI |
| NAME | VARCHAR(45) |  | NN |  |  | The name of the delivery type |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier representing the EEA Role | AI |
| EXT\_ID | CHAR(50) |  |  |  |  | External Unique identifier representing EEA Role |  |

# EEA\_USER

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier representing the EEA User | AI |
| EXT\_USER\_ID | CHAR(55) |  | NN |  |  | External unique identifier representing EEA User |  |

# EVENT

# NOTIFICATION

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the event | AI |
| CHANNEL\_ID | INTEGER |  | NN | UNSIGNED |  | Unique identifier of the referenced channel |  |
| TITLE | VARCHAR(255) |  | NN |  |  | The title of the event |  |
| START\_DATE | DATETIME |  | NN |  |  | A datetime representing when an event starts |  |
| END\_DATE | DATETIME |  |  |  |  | A datetime representing when an event ends |  |
| DESCRIPTION | VARCHAR(255) |  |  |  |  | A description of the event |  |
| CREATION\_DATE | DATETIME |  | NN |  |  | A timestamp representing when an event has been inserted |  |
| CAUSER | CHAR(55) |  |  |  |  | Terse phrase representing the external web service generator of the event |  |
| EXT\_ID | VARCHAR(255) |  | NN |  |  | The external web service unique identifier of the event |  |

# NOTIFICATION\_PROFILE\_TYPE

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the notification |  |
| EEA\_USER\_ID | INTEGER |  | NN | UNSIGNED |  | Unique identifier of the referenced EEA User |  |
| EVENT\_ID | INTEGER |  | NN | UNSIGNED |  | Unique identifier of the referenced event |  |
| CONTENT | TEXT |  | NN |  |  | Message of the notification |  |

# NPT\_DT

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier representing the notification profile type | AI |
| NAME | VARCHAR(25) |  | NN |  |  | The name of the notification profile type |  |
| NOTIFICATION\_TEMPLATE\_ID | INTEGER |  | NN | UNSIGNED |  | Unique identifier representing the referenced notification template |  |

# NPT\_ROLE

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| NOTIFICATION\_PROFILE\_TYPE\_ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the referenced notification profile type |  |
| DILIVERY\_TYPE\_ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the referenced delivery type |  |

# SUBSCRIPTION

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| NOTIFICATION\_PROFILE\_TYPE\_ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the referenced notification profile type |  |
| EEA\_ROLE\_ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the referenced EEA Role |  |

# SUBSCRIPTION\_DT

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the subscription |  |
| EEA\_USER\_ID | INTEGER |  | NN | UNSIGNED |  | Unique identifier of the referenced EEA user |  |
| CHANNEL\_ID | INTEGER |  | NN | UNSIGNED |  | Unique identifier of the referenced channel |  |
| LEAD\_TIME | INTEGER |  |  | UNSIGNED |  | A number representing seconds of the lead time for notification forewarn. |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ColumnName | DataType | Primary  Key | NotNull | Flags | Default Value | Comment | Auto­Inc |
| SUBSCRIPTION\_ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the referenced subscription |  |
| DILIVERY\_TYPE\_ID | INTEGER | PK | NN | UNSIGNED |  | Unique identifier of the referenced delivery type |  |

### RDF schema

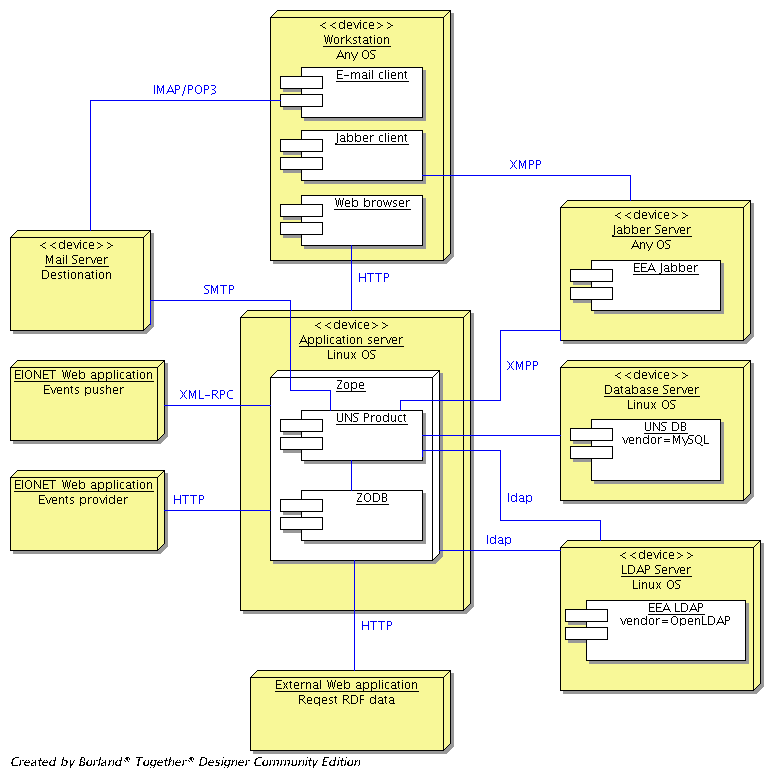
This section aims at providing a detailed description of the RDF schema that will be used for pushing events data from the external web services to the REPORTNET Unified Notification Service.

The table bellow represents some elements of a conceptual schema. Schema structure should be further discussed with EEA in order to produce a final form.

|  |  |
| --- | --- |
| Name | Description |
| Title | Title of the event |
| Start date | Date-time when event starts. |
| End date | Date time when event ends. |
| Source | Reference to the resource within given context. |
| Creation date | Date-time when event has been created in the UNS. |
| Creator | Entity responsible for making event. |
| Related resource | A reference to resource related to the event |

# Deployment view

The REPORTNET Unified Notification Service UML deployment diagram depicts a static view of the run-time configuration of processing nodes and the components that run on these nodes. In other words, the deployment diagrams show the hardware for the UNS, the software that is installed on that hardware, and the middleware used to connect the physically distinct machines to one another.

Illustration 15: The UNS deployment diagram

Application server

It represents a server device which is running under the Red Hat Enterprise Linux. The Zope application server will be installed on this device. Deployment of the REPORTNET Unified Service will take place under this Zope instance. The Zope instance will handle all communication initiated from the outside world. This includes HTTP requests, as well as XML-RPC calls. Zope will communicate with the LDAP Server through the LDAP protocol in order to perform authentication when it is needed. As shown in Figure 15 the UNS product will communicate with the LDAP and Database server.

Database Server

This is a server device running under Red Hat Enterprise Linux. The MySQL database version 4.1 will be installed on this device. The UNS database will be deployed under this MySQL installation.

LDAP Server

This server device represents the existing informational structure of the EEA. It keeps the OpenLDAP server used as credentials storage. The Zope application server will communicate with OpenLDAP in order to perform the authentication procedure. Also the UNS product will use it in order to keep consistency of data between the UNS Database (UNS DB on the figure above) and the EEA LDAP.

**Jabber Server**

This device represents server that runs under any OS with installed Jabber server software. The UNS assumes that it will use existing EEA Jabber server.

**Mail Server**

This device represents server used for mailing purpose from some remote vendor. The UNS will deliver e-mail notifications directly to this destination device.

Workstation

A personal computer used by the users who want access to the UNS or the retrieves notifications. Installed components which will interact with the UNS shall be some internet browsers described in the FSD document. Installed jabber client software and e-mail client software is needed for retrieval of the notifications. Vendors and versions of this software are unimportant for the UNS.

EIONET Web application

It is software that lives outside the UNS. It may be located inside the EEA intranet structure; it may even be some remote resource accessing the UNS through the Internet.

The following list represents two types of the EIONET web applications that may communicate with the UNS:

* *Events pushers*. This type of the EIONET Web applications will push events data to the UNS by using XML-RPC.
* *Events providers*. This type of the EIONET Web applications will provide events data to the UNS in the RSS/RDF format. The UNS will pull this data from this kind of applications.

External web application

It represents software outside the UNS that may require receiving events or notifications data from the UNS. Since the UNS will expose this data as RDF, external web applications will use the HTPP protocol to achieve their goals.

# Appendix A - Interactions diagrams

Interfaces described in section 4 will be used to allow interaction between system modules and interactions between external systems and the UNS.

The following sequence diagrams show these interactions in order accomplish system functions.

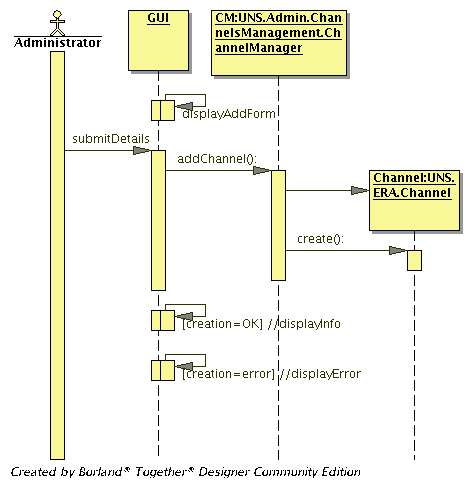
Illustration 16: Create new channel sequence diagram

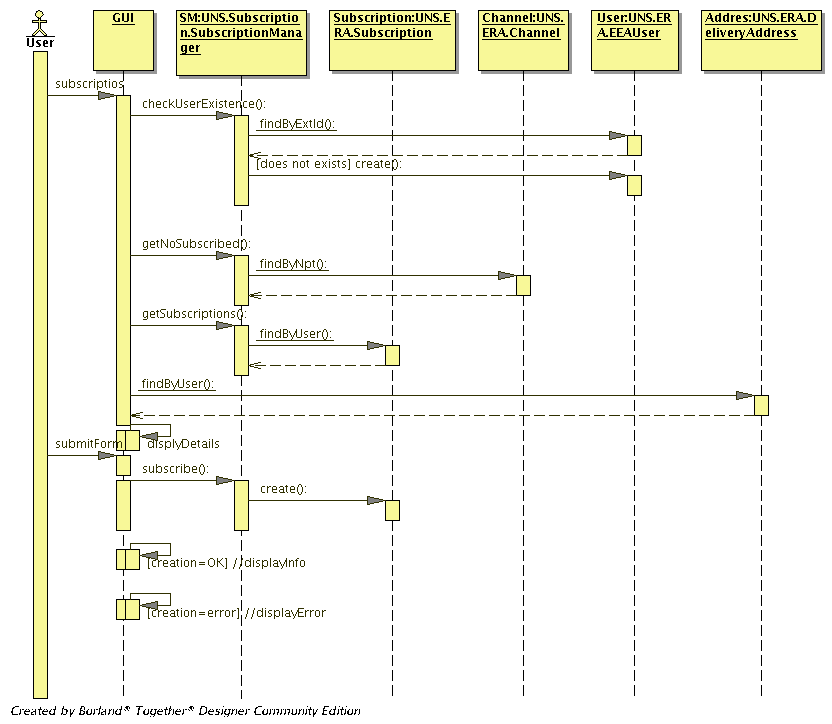
Illustration 17: Subscribe sequence diagram

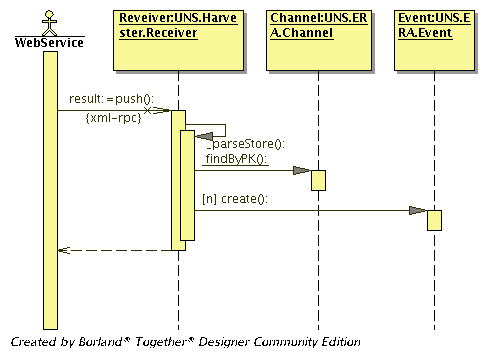
Illustration 18: Pushing events to the UNS sequence diagram

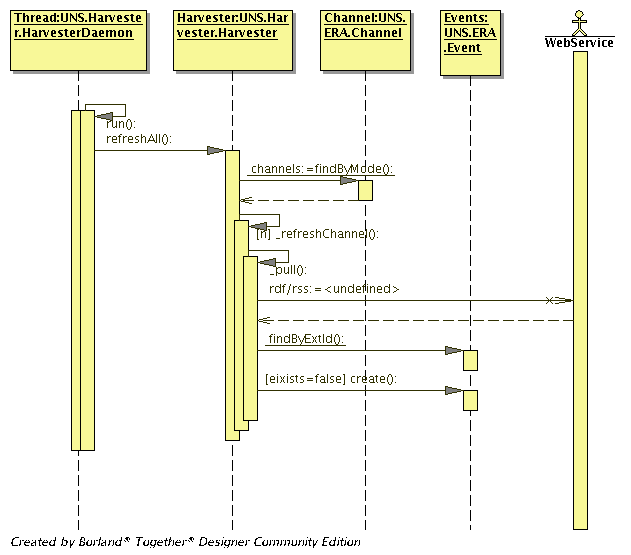
Illustration 19: Harvesting events sequence diagram

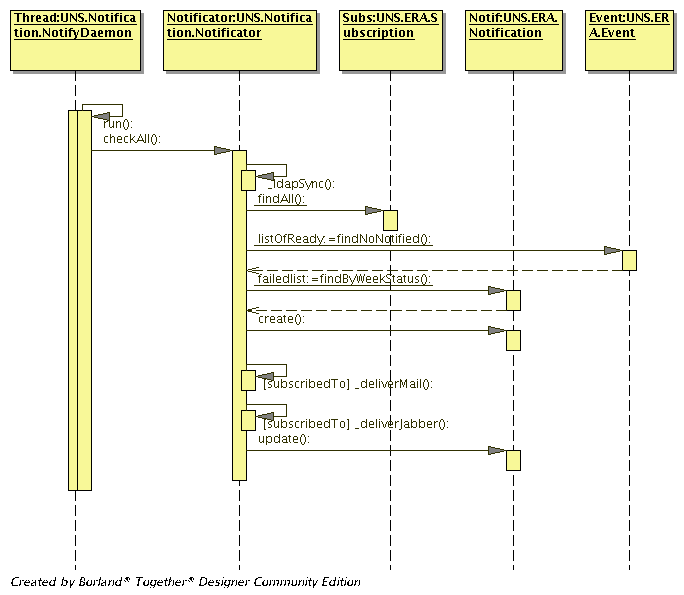
Illustration 20: Sending notifications sequence diagram

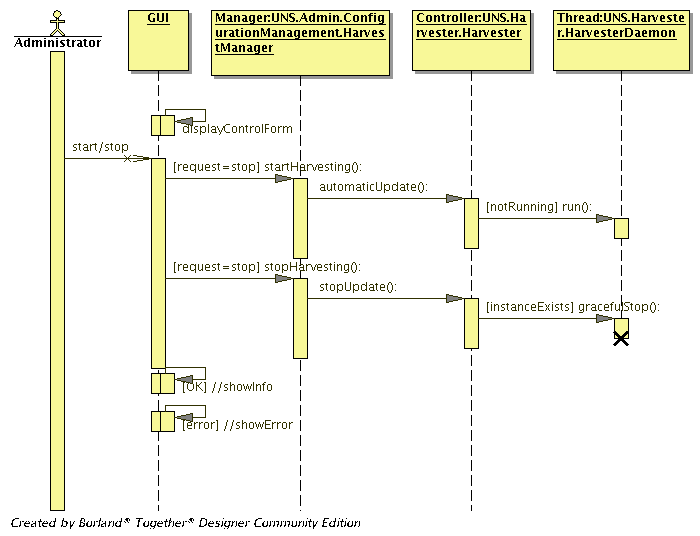
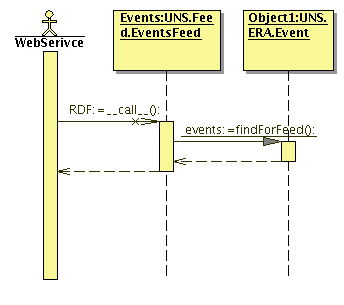
Illustration 21: Start/Stop Harvesting sequence diagram

Illustration 22: Providing events feed to the external web services sequence diagram

1. It is important to stress that the REPORTNET Unified Notification Service will be developed and tested with versions outlined above. [↑](#footnote-ref-2)