Document Type	
EDMS ID Number	Add reference
Date	Add date

Naming Convention Tool Design Document

Author	Affiliation	Editor	Approver
Andraž Požar	Cosylab d.d.		
Marko Kolar	Cosylab d.d.		
Miha Vitorovič	Cosylab d.d.		

European Spallation Source ESS AB
Visiting address: ESS, Tunavägen 24
P.O. Box 176
SE-221 00 Lund
SWEDEN

www.esss.se

DOCUMENT REVISION HISTORY

Version	Reason for revision	Date
1.0	New Document	2014-03-30

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Scope

The scope of this document is description of the design comprising all parts of the tool being developed to support the process of naming devices according to the ESS Naming Convention [2]. This includes

- **Name Administration Tool**: Tools to administer mnemonic name components in the logical area structure and device category structure.
- **Device Naming Tool:** Tool to construct and edit instances of device names according to a set of rules described in the naming convention document [2].

1.2 Definitions, Acronyms and Abbreviations

Item	Description
DISCS	Distributed Information Services for Control Systems
ER	Entity-relationship
ESS	European Spallation Source
FRIB	Facility for Rare Isotope Beam
JPA	Java Persistance API
NC	Naming convention
ORM	Object-relational mapping
ReST	Representational state transfer
NT-[0-9,0-9,0-9]	Requirement number [4]

1.3 Naming Conventions

Names of all devices (beam instrumentation, sensors, actuators, etc.), equipment (power supplies, magnets, RF cavities, targets, moderators, instruments, etc.) in technical systems and conventional facilities in ESS are named using naming convention. This convention is described in details in the ESS naming Convention Document [2].

In summary, all names are constructed from two structures, hierarchically arranged as

- Logical Area Structure Level 1. **Super Section** (SupS)
- Logical Area Structure Level 2. **Section** (Sect)
- Logical Area Structure Level 3*.* **Subsection** (SubS)
- Device Category Structure Level 1. **Discipline** (Dis)
- Device Category Structure Level 2. **Category** (Cat)
- Device Category Structure Level 3. **Generic Device Type** (GDev)

where SupS, Sect, SubS, Dis, Cat and GDev represents the mnemonic name components. These are then put together into a device name

SSSS-BBBB:DDDD-III

Sections are assigned either as the default type (D) or as the alternative (A) type. D-type sections have alphanumeric mnemonic name components, whereas mnemonic name components of subsections under A-type sections are numbers (of string format).

From here on, there are two methods of constructing a device name.

1.3.1 Device names in D-type sections

- **SSSS** – Sect.
- **BBBB** – SubS.
- **DDDD** – GDev.
- **III** – An arbitrary instance index needed to make device name unique.

1.3.2 Device names in A-type sections

- **SSSS** – Sect
- **BBBB** – Dis
- **DDDD** – GDev
- **III** – SubS followed by an optional alphabetic instance index, used for differentiating devices of the same type on the same subsection.

1.4 Syntax Rules

1. Mnemonic names components for any of the items in the logical area structure (SupS, Sect, SubS) and device category structure shall be alphanumeric. I.e., only upper and lower case alphanumeric characters (a-z, A-Z, 0-9) are allowed.
2. First character of mnemonic name components for section Sect, D-type subsection SubS, discipline Dis, generic device type GDev as well as instance index X for A-type devices shall be alphabetic.
3. Mnemonic name components for A-type subsection SubS shall be numeric.
4. Mnemonic name components for section Sect and discipline Dis, specific device types SDev as well as device name SSSS-BBBB:DDDD-IIII shall be unique irrespective of

    (a) Letter case

    (b) Letters I, l and number 1

    (c) Letter O and number 0

   (d) Letters V and W

   (e) Leading zeroes, i.e., number 0 immediately following a non-numerical character

5. Mnemonic name components for subsection SubS and generic device type GDev shall be unique according to rule number 4, however only within the same section and discipline, respectively.

6. Leading zeroes shall be used to ensure that all numbers have the same number of digits.

7. Mnemonic name components shall be composed with suitable and logical use of lower and upper case.

8. The minimum length of mnemonic name component is two characters.

9. The maximum length of mnemonic name component is six characters.

The naming system will support the management of the name mnemonics to support definition of the naming convention in an organized way; authorized users will be able to propose changes to the name mnemonic (addition of a new mnemonic, modification or deletion of an existing one), and the naming administrator will have possibility to accept or reject those changes. The complete history of the mnemonics is retained and can be displayed on request.

2. ROLES AND WORKFLOW

There are three roles involved in the naming process. The name administrators (Adm) administer the data and acts on behalf of the naming committee. The device editors (DE) configure devices and are presumably members of one of the naming sub committees, while the guest (G) is other technical staff. The user privileges are shown in table 2.

Table 1 Privileges to manage items in the logical area structure and device category structure

Task	Adm	DE	G
Browse the name element structures.	yes	yes	yes
Browse device tree.	yes	yes	yes
Search devices in device tree.	yes	yes	yes
Propose new items in the logical area structure and device category structures.	yes	yes	no
Accept or reject proposed items in the logical area and device category structure.	yes	no	no
Propose changes to items in the logical area and device structures	yes	yes	no
Accept or reject proposed changes to items in the logical area and device category structures.	yes	no	no
Mark items to be deleted in the logical area and device category structures.	yes	yes	no
Delete items in logical area and device structures.	yes	no	no
Add instances of device names	yes	yes	no
Edit instances of device names	yes	yes	no
Delete instances of device names	yes	yes	no

Described below is the intended workflow for using the naming tool. It is presented in a form of typical scenarios.

Alice is an engineer who, for whatever reason, would like to connect an espresso machine to the control system. In this context Alice is our device editor.

1. Browse and search the device category and logical area structures:
   1.1. Alice opens the Name administration tool and browses the device category structure to search for a generic device type.
   1.2. She looks under the discipline controls (Ctrl) but cannot find a generic device type for the espresso machine.
   1.3. She decides to add espresso machine to the list of generic device types.
2. Add a new generic device type (GDev):
   2.1. Alice selects Controls (Ctrl) => Miscellaneous (Misc) from the tree of device categories.
   2.2. Alice clicks the add button to add a generic device type under the category Miscellaneous (Misc).

2.3. In the field labeled Full Name she enters «Espresso Machine».

2.4. She enters «EspMch» in the mnemonic name field. The proposed name is acceptable, since the maximum length for mnemonics is 6 characters.

2.5. Alice adds a comment «Device to prepare espresso.» in the comment field.

2.6. Alice clicks the submit button. As a device editor Alice can only propose new generic device types. To make the generic device type active it has to be approved by a super user Bob. When Bob logs into the application he can see all the proposed changes marked with different colors in the view where all names are listed.

3. Approve or reject a requested device name

3.1. When Bob logs into the application he can see all the proposed changes in the view where all names are listed.

3.2. Bob selects proposed change and decides to approve the proposal.

3.3. Into the message field, Bob enters «Sure, who doesn’t need a control system to control a coffee machine.»

3.4. Bob presses the finish button. Alice is able to see the approval in the history of the name mnemonic.

4. Reserve device name.

4.1. Alice opens up the device naming tool

4.2. She selects Head Quarter (HQ) ➔ Integrated Control Center (ICC) ➔ Main Control Room (MCR) from the Area breakdown tree view.

4.3. She presses the add button to add a new device to the naming database

4.4. Alice selects the device type from the device category tree, guided by the question «What kind of device is it?» Alice selects Controls (Ctrl) ➔ Miscellaneous (Misc) ➔ Espresso Machine (EspMch).

4.5. Alice adds the correct instance index into the instance index field and presses the submit button. A naming convention name, with correct quantifier and everything, is created.

3. DATABASE DESIGN

PostgreSQL will be used as the database management system for the naming tool's schema which is the recommended technology for ESS software development [1]. The database schema is automatically generated from the object model classes by the JPA layer.

3.1 Database Diagrams

3.1.1 Naming Convention Tool Database Diagram

In the ER diagram below are the tables that are used by the Naming tool. Foreign key references are presented as arrows between tables.

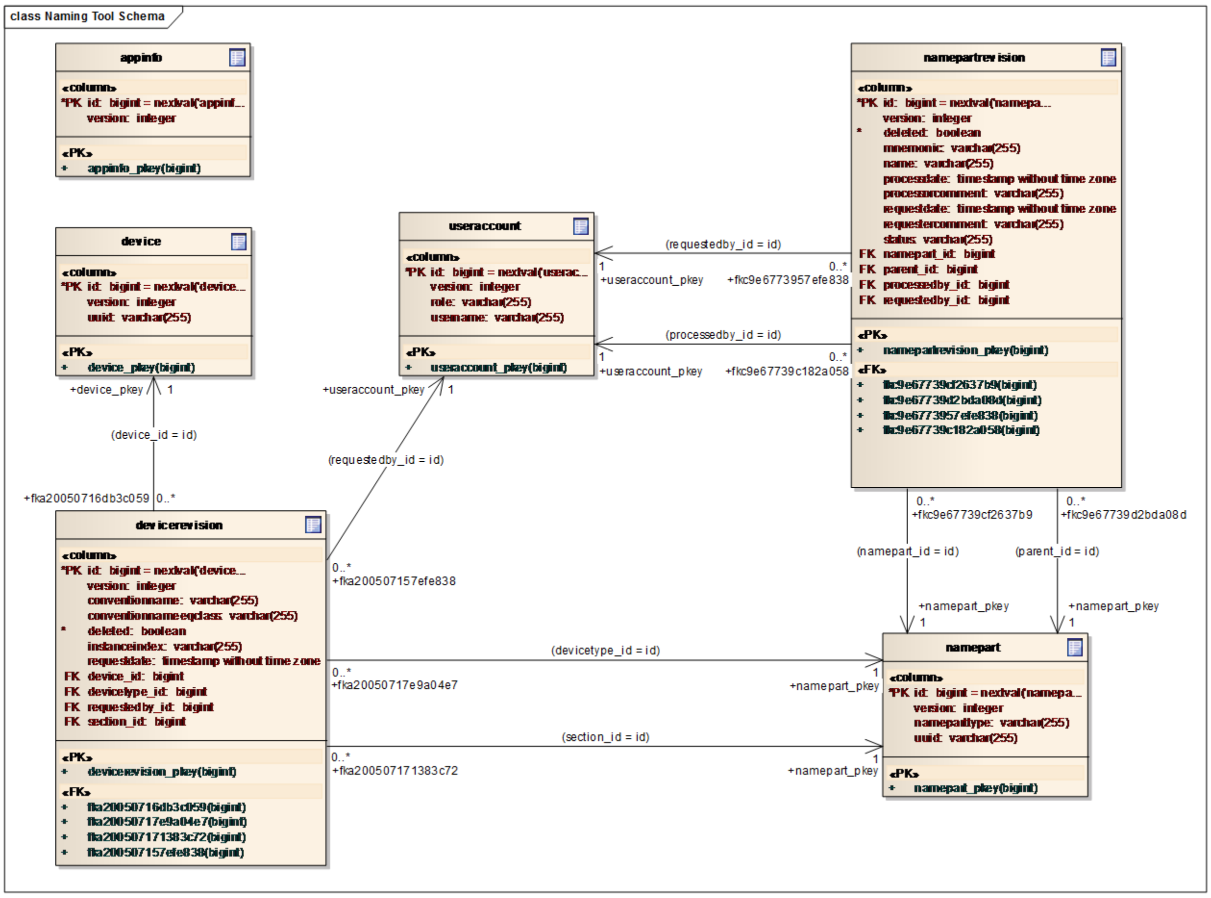

Figure 1 ER model diagram of the Naming Tool database

3.2 Entity Tables and Concepts

All tables will have corresponding persistence entities (shown in Figure 3) that will be part of the Java API for the application. Persistence entities will be defined and described using JPA annotations.

Tables list all the columns and their descriptions of corresponding SQL tables.

3.2.1 namepart

FIELD	DESCRIPTION
`id`	Unique ID for the mnemonic entity.
`uuid`	Universally unique identifier of the mnemonic.
`nameparttype`	Type of a name part specifying whether it belongs to the Logical Area Structure or the Device Category Structure.
`version`	Internal JPA field.

`namepart` table stores the different mnemonics and is an entry point for the revision history of this mnemonic. The field UUID is the unchangeable unique identifier of the mnemonic and is referenced by other database entities, e.g. other mnemonics that are below it in the hierarchy, or the name instances. Even if the mnemonic changes its UUID remains the same.

3.2.2 namepartrevision

FIELD	DESCRIPTION
`id`	Unique ID for the mnemonic revision entitiy.
`mnemonic`	The short, mnemonic name of the part in accordance with the naming convention
`name`	The long, descriptive name of the part. Does not need to follow a convention.
`namepart_id`	Reference to the `namepart` entity which this revision is a part of.
`parent_id`	The parent of this name part in the hierarchy. Null if at the top of the hierarchy.
`status`	Status of the mnemonic revision (pending/approved/rejected/canceled)
`deleted`	A flag signifying that the revision represents deletion of the name part
`requestedby_id`	The user that proposed the revision. Null if the revision was generated by an automated process.
`requestdate`	Timestamp of the name request creation.
`requestercomment`	The comment the user gave when proposing the revision. Null if no comment was given.
`processedby_id`	The user who processed the revision. Null if the revision was processed by an automated process.
`processdate`	Timestamp of the name event processor.
`processorcomment`	The comment the administrator user gave when processing the revision. Null if no comment was given.
`version`	Internal JPA field.

namepartrevision table holds all the names of the specific name category instances. All names have a full name and also a naming convention mnemonic that must be unique. These mnemonics will be used when constructing the NC names.

parent_id column is used to reference another namepart entry. This allows rendering namepart entries (mnemonics) into tree-like hierarchical structures within the table {NT-003, NT-015}.

namepartrevision also stores data used for name audits {NT-006,NT- 012}, such as: who requested the name to be added, changed {NT-005, NT-009NT-048, NT-050} or deleted, when did this happen, what was the reason for the change, who and when processed this request, etc.

3.2.3 device

FIELD	DESCRIPTION
id	Unique ID for the name instance entity.
UUID	Universally unique identifier of the device name.
version	Internal JPA field.

device entries represent name instances for individual devices. The table is an entry point for a revision history of the name instances. All entities that must reference a name instance, reference a name instance identified by UUID. This way the correct name instance is always referenced regardless of its history or its current value.

3.2.4 devicerevision

FIELD	DESCRIPTION
id	Unique ID for the name instance revision entity.
device_id	Reference to the to the device entity which this revision is a part of.
section_id	The section containing the device.
devicetype_id	The type of the device.
instanceindex	An additional identifier that, in combination with other attributes, determine the unique.
deleted	A flag signifying that the revision represents deletion of the device.
requestdate	The time when the revision was proposed.
requestedby_id	The user that proposed the revision. Null if the revision was generated by an automated process.
conventionname	The full name of the device in accordance with the naming convention.
conventionnameeqclass	The representative of the equivalence class the convention

FIELD	DESCRIPTION
	name belongs to. This is used to ensure uniqueness of convention names when treating similar looking names (for example, containing 0 vs. O, 1 vs. l) as equal.
`Version`	Internal JPA field.

Entries in the `devicerevision` table represent a revision of the name instance and hold references to all mnemonics needed to uniquely identify them {NT-013, NT-015, NT-023, NT-025}. Number of different references restricts the number of independent naming part hierarchies that can be used in the construction of a full name. At the moment there are two such references:

- `section_id`, referring to the part of the facility the item with this name will provide service to.
- `devicetype_id`, referring to the discipline or type of the item with this name.

In addition to this, there is also an `instanceindex` field (previously quantifier) used for differentiating devices of the same specific type on the same subsection.

Normalized convention name is stored in the table to provide quick check for uniqueness when adding new device names or changing old one.

3.2.5 useraccount

FIELD	DESCRIPTION
`id`	Unique ID of the user's account entity.
`username`	The name identifying the user.
`role`	The role that determines the user's access control permissions.
`version`	Internal JPA field.

`useraccount` entity holds basic information about user's role. It holds user's `username` and his role. Different user roles are:

- Name Administrator – full editing permissions and permissions to approve/reject proposed changes {NT-005, NT-009, NT-052}.
- Device editors – permissions to propose modifications/deletion/addition of logical area structure and device category structure. Has permissions to add/modify/delete device names without administrators approval {NT-014, NT-048, NT-050}.

- Guests – read only permissions.

Device editors are planned to have permissions to freely add/modify/delete all device names and to propose changes to name parts. User privileges are listed in table 1.

4. APPLICATION ARCHITECTURE AND DESIGN

4.1 Overview

The application is built of cleanly separated layers, with each layer in contact only with the two layers immediately above and below.

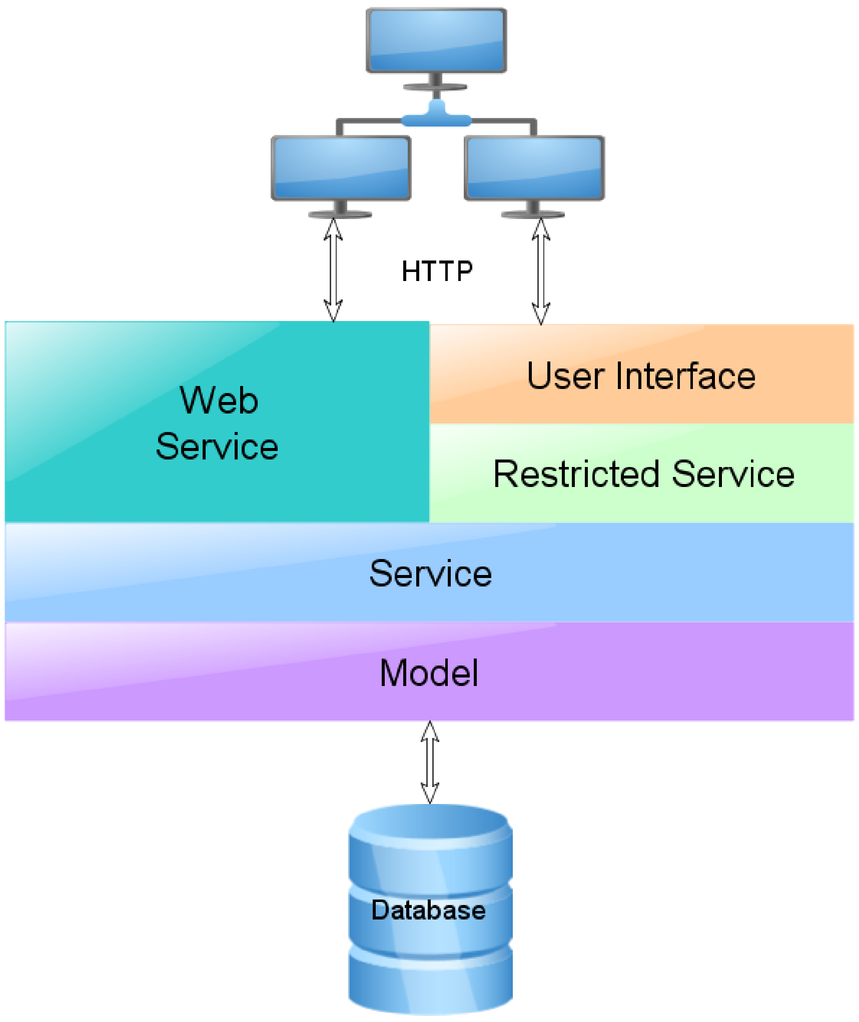

Figure 2 Architectural overview

4.2 Model

The model is located in `org.openepics.names.model`. It's implemented as a set of JPA classes, all extending a common base class `Persistable`, which defines the standard JPA fields.

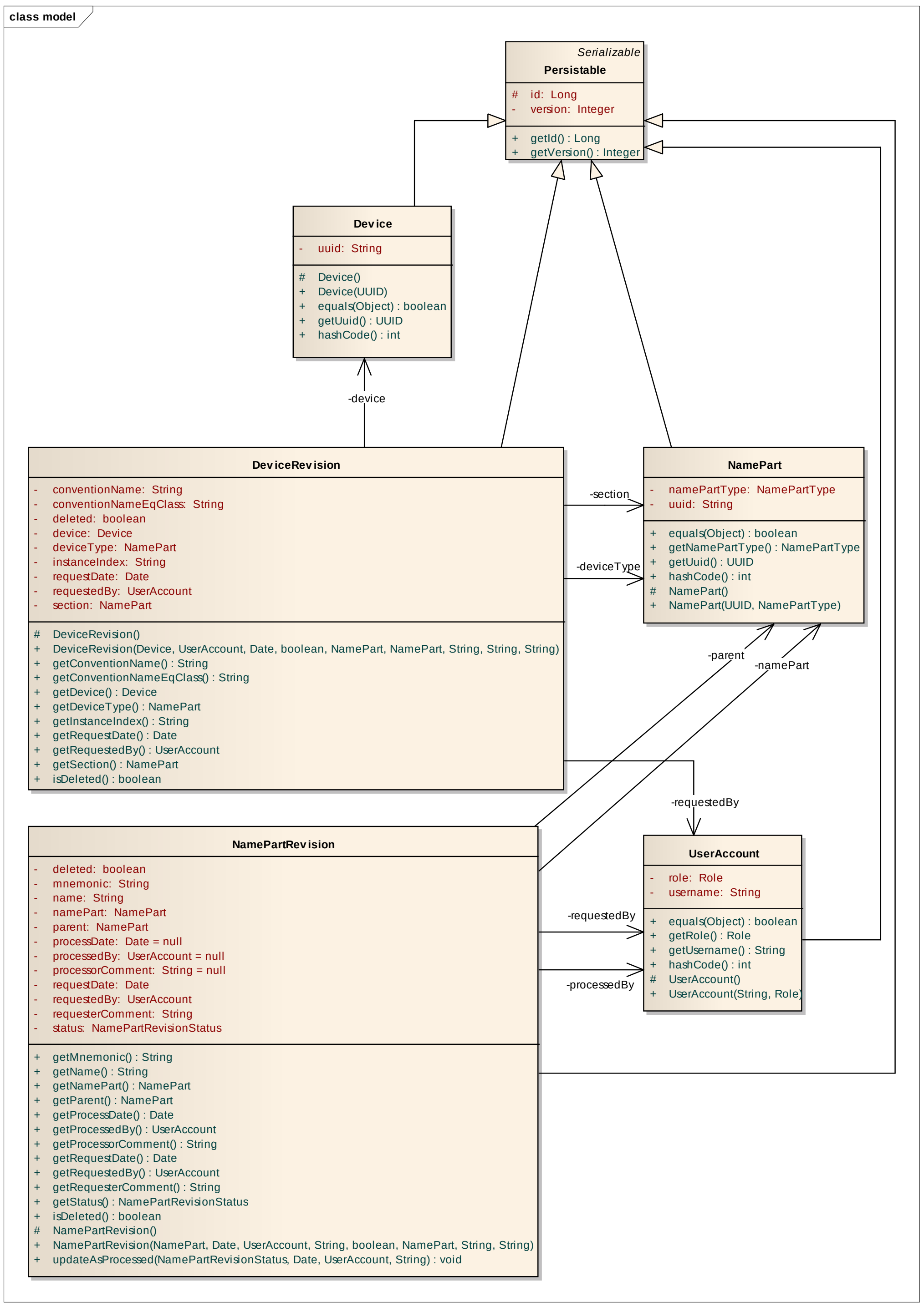
class model
Serializable
Persistable
# id: Long
- version: Integer
+ getId() : Long
+ getVersion() : Integer
Device
- uuid: String
# Device()
+ Device(UUID)
+ equals(Object) : boolean
+ getUuid() : UUID
+ hashCode() : int
-device
DeviceRevision
- conventionName: String
- conventionNameEqClass: String
- deleted: boolean
- device: Device
- deviceType: NamePart
- instanceIndex: String
- requestDate: Date
- requestedBy: UserAccount
- section: NamePart
# DeviceRevision()
+ DeviceRevision(Device, UserAccount, Date, boolean, NamePart, NamePart, String, String, String)
+ getConventionName() : String
+ getConventionNameEqClass() : String
+ getDevice() : Device
+ getDeviceType() : NamePart
+ getInstanceIndex() : String
+ getRequestDate() : Date
+ getRequestedBy() : UserAccount
+ getSection() : NamePart
+ isDeleted() : boolean
-section
-deviceType
NamePart
- namePartType: NamePartType
- uuid: String
+ equals(Object) : boolean
+ getNamePartType() : NamePartType
+ getUuid() : UUID
+ hashCode() : int
# NamePart()
+ NamePart(UUID, NamePartType)
-parent
-namePart
-requestedBy
NamePartRevision
- deleted: boolean
- mnemonic: String
- name: String
- namePart: NamePart
- parent: NamePart
- processDate: Date = null
- processedBy: UserAccount = null
- processorComment: String = null
- requestDate: Date
- requestedBy: UserAccount
- requesterComment: String
- status: NamePartRevisionStatus
+ getMnemonic() : String
+ getName() : String
+ getNamePart() : NamePart
+ getParent() : NamePart
+ getProcessDate() : Date
+ getProcessedBy() : UserAccount
+ getProcessorComment() : String
+ getRequestDate() : Date
+ getRequestedBy() : UserAccount
+ getRequesterComment() : String
+ getStatus() : NamePartRevisionStatus
+ isDeleted() : boolean
# NamePartRevision()
+ NamePartRevision(NamePart, Date, UserAccount, String, boolean, NamePart, String, String)
+ updateAsProcessed(NamePartRevisionStatus, Date, UserAccount, String) : void
-requestedBy
-processedBy
UserAccount
- role: Role
- username: String
+ equals(Object) : boolean
+ getRole() : Role
+ getUsername() : String
+ hashCode() : int
# UserAccount()
+ UserAccount(String, Role)

Figure 3 Naming Tool ORM class diagram

The model classes also serve as the authoritative definition of the database schema, which is automatically generated from them by JPA layer.

Model contains the following classes:

Table 2 Classes of the org.openepics.names.model package

Class	Description
`AppInfo`	A singleton entity representing the installed Naming Tool application.
`Device`	An entity representing a device.
`DeviceRevision`	A revision of a `Device` entity representing its state at some point in time.
`NamePart`	An entity representing either a named section of the Logical Area Structure or a named device type of the Device Category Structure, depending on the specified `NamePartType`.
`NamePartRevision`	A revision of a `NamePart` entity representing its state at some point in time.
`Persistable`	A superclass implementing the properties required by JPA.
`UserAccount`	An entity representing a user account used to sign in to the application.

Table 3 Enums of the org.openepics.names.model package

Enum	Description
`NamePartRevisionStatus`	Status of the name part in the request / approve workflow.
`NamePartType`	Type of a `NamePart` specifying whether it belongs to the Logical Area Structure or the Device Category Structure.
`Role`	`UserAccount` role that determines the user's access control permissions.

4.2.1 Revision control

Some model classes (`NamePart` and `Device`) implement a revision control scheme, which allows the application to keep the full history of changes made to the entities, instead of overwriting their properties on each change.

The scheme is implemented with a pair of classes. The first one, for example `NamePart`, represents the entity irrespective of its changing state and only carries a single field with a Universally unique identifier (`UUID`).

The second class, for example `NamePartRevision` represents the state of the entity's properties at some point in time. In addition to the property values, this class contain fields with meta-information on when and by whom the revision was created as well as an optional comment.

The revision objects reference their entity by a relational field. They are ordered chronologically by their JPA ids.

4.2.2 Request / approve workflow

To support a workflow where a user can request a change and another user reviews and approves the change, some revision classes also contain meta-information fields on the status of the revision (pending, approved, rejected or cancelled) and fields on when and by whom the revision was reviewed and an optional comment of the reviewer.

4.3 Service

The service layer is located in `org.openepics.names.services.` It's implemented as a set of Java EE beans with methods implementing queries and operations on the model entities.

The operations are designed to be atomic, and to always result in a globally consistent state. In this regard special care was taken with the logic related to the request / approve workflow, which ensures that at to time can the database contain pending requests for changes that conflict one another.

The operations are called on a single entity. When the user wants to act on multiple entities at the same time, they can be composed by calling them sequentially in a single transaction. The order is important for the successful completion of the transaction, but an incorrect order will not result in an inconsistent state, the transaction will simply fail. Selection of the correct operation order based on multiple selection is described in the User Interface section.

Table 4 Classes of the org.openepics.names.services package

Class	Description
`ApplicationService`	A service bean managing global application settings and database initialization.
`EssNamingConvention`	A naming convention definition used by ESS.
`FribNamingConvention`	An empty stub for the naming convention used by FRIB.
`InitialDataImportService`	A service bean used to initialize the database with data from the bundled Excel file when the application is ran for the first time.
`NamePartService`	A service bean managing `NamePart` and `Device` entities.
`SessionService`	A session bean holding the `UserAccount` entity representing the signed in user.
`UserService`	A service bean managing `UserAccount` entities.

4.3.1 Views

With a model that supports revision control it can be difficult to answer questions like "what is the name of this entity's hierarchical parent?". We cannot answer such questions just by looking at the single object's fields, because in addition to finding the parent entity, we need to look up its current revision which then carries the name.

To make it possible to answer such relational questions in a more object-oriented way, for example just by calling `entity.getParent().getName()`, we first wrap the entity in a `View`, which exposes these methods. In the background the `View` is connected to a service bean and will transparently run the appropriate database queries when we look at its properties.

Table 5 Classes of the org.openepics.names.services.views package

Class	Description
`DeviceView`	A view of a `Device` (its particular revision) that makes it easy to query some of its properties and relations in an object-related fashion.
`NamePartView`	A view of a `NamePart` that makes it easy to query some of its properties and relations in an object-related fashion.

4.3.2 Naming Convention

The application is designed to be usable by different facilities, each with their own naming convention and rules. A new naming convention can easily be defined by writing a class that implements the `org.openepics.names.services.NamingConvention` interface, and configuring it in `beans.xml`.

The interface defines methods for name validation, defining equivalence between names, and for composing a full device name from the named (sub)sections and (sub)types that define a device.

Table 6 Naming convention interface

Interface	Description
`NamingConvention`	An interface defining the naming convention to be used by the application that includes: • name validation rules • name uniqueness rules • form of composite names The used naming convention is configured through beans.xml using the CDI alternatives mechanism.

4.4 Restricted Service

The service layer does not concern itself with what user is calling it and whether that user has the appropriate access permissions, only that the operation itself is legal.

This is instead handled by a separate *Restricted Service layer* located in `org.openepics.names.services.restricted`, which serves as a gateway to unrestricted services and checks the user permissions before executing the operation or query.

This design allows for clean separation of concerns and makes it possible for different automated services to call each other without hacks for circumventing restrictions meant for real, human users.

For example, the *Web Service layer*, which is meant to connect different application modules which implement their own access control, connects directly to the unrestricted service layer. The *User Interface layer*, on the other hand, should always go strictly through the *Restricted Service layer*.

4.5 User Interface

The user interface implemented with JSF. The pages (screens) are defined in `xhtml` *files* found under the `webapp` folder. Each page is paired with a view-scoped controller located in `org.openepics.names.ui` that holds the UI state and implements the more complex aspects of the UI.

Table 7 Classes of the org.openepics.names.ui.devices package

Class	Description
`AddInstanceIndexValidator`	The validator for the *Instance index* field in the *Add form*.
`DevicesController`	A UI controller bean for the *Device Names* screen.
`DevicesTreeBuilder`	Utility bean for building JSF `TreeNode` trees from the section hierarchy and containing devices as leaf nodes.
`ExcelImport`	A bean for importing devices from Excel.
`ModifyInstanceIndexValidator`	The validator for the *Instance index* field in the *Modify form*.

Table 8 Classes of the org.openepics.names.ui.parts package

Class	Description
AddMnemonicValidator	The validator for the *Mnemonic* field in the *Add form*.
ModifyMnemonicValidator	The validator for the *Mnemonic* field in the *Modify form*.
NamePartsController	A UI controller bean for the Logical Area Structure and Device Category Structure screens.
NamePartTreeBuilder	A utility bean for building JSF `TreeNode` trees from the `NamePart` hierarchy.

Table 9 Classes of the org.openepics.names.ui.export package

Class	Description
`ExcelExport`	A bean for exporting sections, device types and devices to Excel.

Because most of the UI deals with manipulating hierarchical data, we use a common set of helper classes and patterns for working with trees. The common classes are found under `org.openepics.names.ui.common`.

Table 10 Classes of the org.openepics.names.ui.common package

Class	Description
`AlphanumComparator`	The Alphanum Algorithm is an improved sorting algorithm for strings containing numbers. Instead of sorting numbers in ASCII order like a standard sort, this algorithm sorts numbers in numeric order.
`CustomExceptionHandler`	A global JSF exception handler that displays caught exceptions in the UI as popup messages.
`CustomExceptionHandlerFactory`	A factory for the `CustomExceptionHandler`.
`LoginController`	A UI controller bean for the login / logout button and form.
`OperationTreeGenerator<T>`	A generator that takes a `TreeNode` tree as input and, based on the selected nodes, generates a new tree with additional information on what nodes will be affected by an operation.
`OperationView<T>`	A wrapper for `TreeNode` data with additional information on whether the node is affected by an operation.
`PreferencesController`	A UI controller bean for the Preferences screen.
`TreeFilter<T>`	A filter that transforms a `TreeNode` tree into a new filtered tree based on acceptance criteria for node's data.
`UserManager`	A bean exposing the information about the logged in user to the UI.
`ViewFactory`	A factory bean for creating augmented views of various entities.

4.5.1 Building the trees

To reduce the number of database queries when first loading hierarchical data for display, we fetch the set of relevant entities and revisions in a single query and then arrange them in a tree in memory. This is done by the `NamePartTreeBuilder` and `DevicesTreeBuilder` classes.

The resulting trees hold the aforementioned `View` classes as node data which make it easy to bind the data for display.

4.5.2 Filtering

Once the original tree is built, it’s kept in memory for the duration of the user’s interaction with the page. The user can then apply different filter criteria to reduce the amount of displayed.

This filtering is done using the `TreeFilter` class which can be given simple criteria whether to keep or omit a node and will automatically cull subtrees with no kept nodes, while keeping parents of kept nodes for context.

4.5.3 Operations

The user can then perform various operations on the selected tree nodes. Some operations make sense only on single nodes, while others can be done on multiple selected nodes. Some operations should also be applied recursively to all descendant nodes when a node is selected.

In each case we need to show a list of available actions to the user. When the user calls an action it's also a good idea to present them with a visual summary of what nodes will be affected by the operation for confirmation.

Finally, the affected set of entities needs to be serialized with the correct order with which to call the operations in the service layer.

All these problems are solved simultaneously using the `OperationTreeGenerator` class. The class is configured with criteria on whether the operation can affect a selected entity, whether to implicitly select its children, or not to affect the children even when they are selected along with the parent.

The `OperationTreeGenerator` is then applied to the tree with the user's selections and produces a tree augmented with additional information on what entities will be affected by the operation.

This tree is immediately usable to display as a preview of the operation's effect. At the same time we can tell if the action itself is possible based on whether or not the tree is empty, because the generator will automatically cull parts of the tree with no affected nodes.

When the user confirms the action, the same tree can be walked recursively to produce an ordered list of operands.

4.6 Web service

The web service is implemented using the standard JAX-RS annotated classes. It connects directly to the Service layer, bypassing user access controls.

The rest service is implemented in the package `org.openepics.names.webservice` which contains 4 classes (Figure 4).

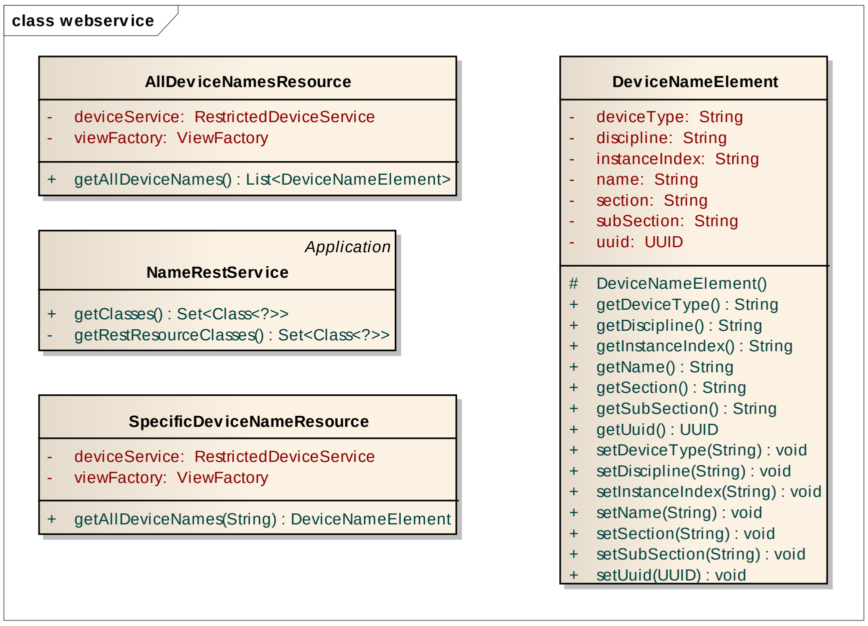
class webservice
AllDeviceNamesResource
- deviceService: RestrictedDeviceService
- viewFactory: ViewFactory
+ getAllDeviceNames() : List<DeviceNameElement>
Application
NameRestService
+ getClasses() : Set<Class<?>>
- getRestResourceClasses() : Set<Class<?>>
SpecificDeviceNameResource
- deviceService: RestrictedDeviceService
- viewFactory: ViewFactory
+ getAllDeviceNames(String) : DeviceNameElement
DeviceNameElement
- deviceType: String
- discipline: String
- instanceIndex: String
- name: String
- section: String
- subSection: String
- uuid: UUID
# DeviceNameElement()
+ getDeviceType() : String
+ getDiscipline() : String
+ getInstanceIndex() : String
+ getName() : String
+ getSection() : String
+ getSubSection() : String
+ getUuid() : UUID
+ setDeviceType(String) : void
+ setDiscipline(String) : void
+ setInstanceIndex(String) : void
+ setName(String) : void
+ setSection(String) : void
+ setSubSection(String) : void
+ setUuid(UUID) : void

Figure 4 Classes implementing ReSTful service

The class `NameRestService` is used to define the components of a ReSTful Web service application deployment and provide additional metadata. It also defines the ReSTful web service context path using the `javax.ws.rs.ApplicationPath` annotation.

5. REST SERVICES

The ReSTful interface offers two methods for obtaining the registered names. The first one returns a list of all registered device instance names, and the second one returns detailed information about a specific name, if such name exists. The root URL of all ReSTful services is at `http://<server>/names/rest/` URL.

5.1 List of all registered device names

The service for returning a list of all registered device names is implemented in the `AllDeviceNamesResource` class and is located at `http://<server>/names/rest/deviceNames` URL. It returns a list of all registered names and name details as XML or JSON. This ReSTful service implements only the GET method.

The example shows the XML response to service URL request:

```
<collection>
  <deviceNameElement>
    <deviceType>Mdl</deviceType>
    <discipline>EMR</discipline>
    <name>RFQ-EMR:Mdl-01</name>
    <section>RFQ</section>
    <subSection>01</subSection>
    <uuid>eb4cde9c-4e6e-41e9-a716-2fa13aa44fe5</uuid>
  </deviceNameElement>
  <deviceNameElement>
    <deviceType>BPM</deviceType>
    <discipline>PBI</discipline>
    <instanceIndex>a</instanceIndex>
    <name>MEBT-PBI:BPM-01a</name>
    <section>MEBT</section>
    <subSection>01</subSection>
    <uuid>d3e991bd-9cea-4950-bc0a-8f8f11b5d915</uuid>
  </deviceNameElement>
  …
</collection>
```

5.2 Details about specific name

The service for returning details about a specific registered name is implemented in the `SpecificDeviceNameResource` class and is located at `http://<server>/names/rest/deviceNames/<UUId>` URL. It returns details about a specific name identified by its id as XML or JSON. This ReSTful service implements only the GET method.

The example shows the JSON response to the following request `http://<server>/names/rest/deviceNames/eb4cde9c-4e6e-41e9-a716-2fa13aa44fe5`:

```
{
    "uuid": "eb4cde9c-4e6e-41e9-a716-2fa13aa44fe5",
    "section": "RFQ",
    "subSection": "01",
    "discipline": "EMR",
    "deviceType": "Mdl",
    "instanceIndex": null,
    "name": "RFQ-EMR:Mdl-01"
}
```

6. REFERENCES

[1] Recommended Technologies for Software Development (https://plone.esss.lu.se/ics/software-core-components/bled/recommended-technologies-for-sw-dev/at_download/file)

[2] ESS Naming Convention Document (http://eval.esss.lu.se/DocDB/0000/000004/010/ESS%20Naming%20convention_20131011_v2.pdf)

[3] DISCS Names Design Document (https://github.com/openepics/names/raw/master/docs/Names-Design-v2.3.pdf)

[4] Naming Tool Requirements (https://plone.esss.lu.se/ics/software-core-components/wu-3-3-naming-convention/naming-requirements/file_download_version?version_id=4)

[5] DISCS Names GIT Repository (https://github.com/openepics/names)