TEAMCENTER

Advanced Classification — Deployment and Administration

Teamcenter 2412

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a global leader in the growing field of product lifecycle management (PLM), manufacturing operations management (MOM), and electronic design automation (EDA) software, hardware, and services. Siemens works with more than 100,000 customers, leading the digitalization of their planning and manufacturing processes. At Siemens Digital Industries Software, we blur the boundaries between industry domains by integrating the virtual and physical, hardware and software, design and manufacturing worlds. With the rapid pace of innovation, digitalization is no longer tomorrow’s idea. We take what the future promises tomorrow and make it real for our customers today. Where today meets tomorrow. Our culture encourages creativity, welcomes fresh thinking and focuses on growth, so our people, our business, and our customers can achieve their full potential.

Support Center: **support.sw.siemens.com**

Send Feedback on Documentation: **support.sw.siemens.com/doc_feedback_form**

Contents

Administering classification

1. Deploying and administering classification

Teamcenter Classification helps to manage and classify product data efficiently.

As an installer, you can install and setup classification for the business users and classification administrator

As a classification administrator, you can create a classification hierarchy containing classes and class attributes that helps you describe the objects that you want to reuse in your organization. Alternatively, you can import a standardized ECLASS hierarchy to describe your data. The classification administrator sets up such features as the sharing of hierarchy and classified data, adding images to classes, or enabling graphics creation for part family templates.

Where do I go from here?

 Business User	You can use advanced classification for reuse and standardization. Alternatively, you can also use the old version of classification using the rich client.
 Installer	
Plan the deployment of classification	**Planning the deployment of advanced classification**
Learn which components must be installed for various classification scenarios	**Install classification**
 Classification administrator	
You use classification in the Teamcenter rich client and now want to search for and author classification data in Active Workspace.	Configure full-feature classification for traditional classification data
You are just beginning with classification and want to create hierarchies so that you can easily share your data.	**Configure classification standard taxonomy**
You want to work with basic and advanced classification simultaneously.	**Classify with advanced and basic classification simultaneously**
You use the ECLASS standard for classifying data and want to download and use such hierarchies and data.	**Configure classification for ECLASS compliant data**
How do you create classification hierarchy for basic and advanced classification data?	**Refer to authoring classification hierarchy objects in Classification Manager**

2. Overview of advanced classification, ECLASS, and licensing

Advanced classification is based on the *classification standard taxonomy* (CST) framework. This framework supports using standardized classification hierarchies and data so that classification data can easily be shared between customers and suppliers. In particular, the framework supports the ECLASS standard that describes products and services across a wide range of industries through the use of classes and properties with unique identifiers.

For more information about the ECLASS standard, visit their website:

https://eclass.eu/en/eclass-standard/introduction

For information about what versions of ECLASS are supported, see ECLASS version support for the current version of Teamcenter.

Teamcenter Classification for ECLASS introduces CST to support ECLASS structures. CST classes are available in the classification hierarchy, and if desired, alongside traditional classification classes (ICS) to classify your data. Using Teamcenter Classification for ECLASS, you can import and export ECLASS-compliant data.

The use of the ECLASS standard requires registration and licensing.

Advanced Classification can also be used as generic classification system where you can build your own hierarchy and classification definitions by using the new features and data constructs. Most of the new features and data constructs are supported in Advanced Classification even without ECLASS.

If you want standards based classification system then ECLASS Standard is supported in Advanced Classification by importing ECLASS taxonomy and classification definitions.

Classification standard taxonomy features (CST) are available with a regular classification authoring license. Reading, authoring, and browsing of a custom CST hierarchy is possible.

If, however, you download and use hierarchy data from ECLASS, you require a **cls_eclass_user** named user license. With this license, you can perform create, update, and delete operations on an ECLASS hierarchy and data.

For complete information about licenses required, contact your Siemens Digital Industries Software representative.

3. What is the difference between basic and advanced classification?

When using the classification feature, there are two types of data that can exist in your classification hierarchy — Basic and Advanced. These are some of the differences between these two types of data:

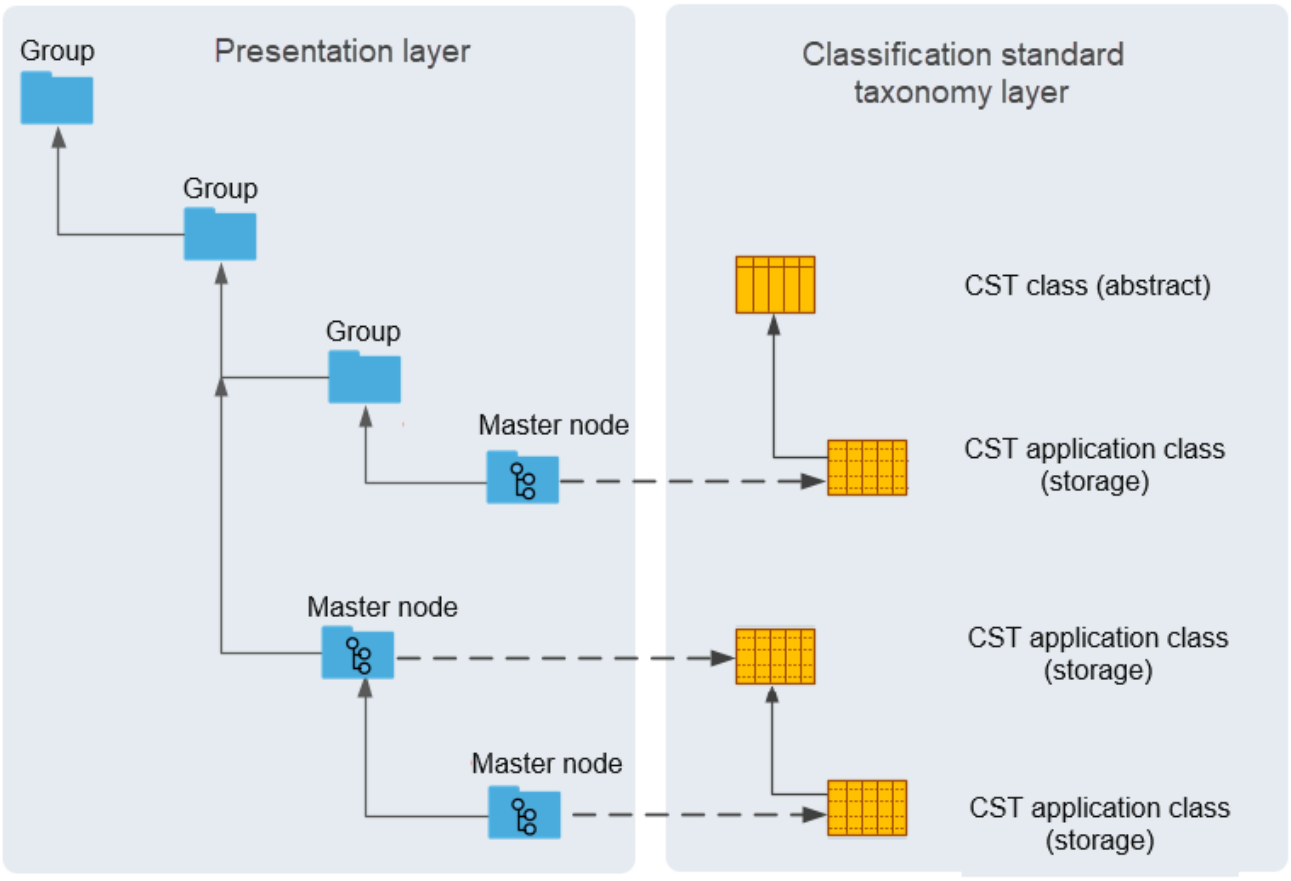
	Basic classification	Advanced classification
Availability	Available on the rich client and Active Workspace.	Available on Active Workspace.
Effectiveness	Effective in defining objects uniquely with properties for reuse.	Effective in capturing overall product information for PIM and MDM systems with support for ECLASS standard class hierarchy and definitions.
Standard Features	Some of the basic classification features: • Classification hierarchical representation • Flat list of properties • List of values • Unit of measure • Support for views • Limited to 200 Properties in a class	Some of the advanced classification features: • Class definition versioning • Flexible data modeling with attribute blocks, aspects, cardinality, and polymorphism. • Complex data type attributes • Native data storage • Support for namespaces • Unlimited number of properties • No Array limitations
Data model standards	Supports DIN-4000 standard and confirms to underlying specifications of ISO/TS-13399 standard.	Supports ECLASS standard and conforms to DIN 4002, ISO 13584-32, ISO 13584-42, IEC 61360 standards.
Hierarchy and class definitions	Define custom hierarchy and class definitions.	Define custom hierarchy and class definitions. Additionally supports standard based ECLASS hierarchy spanning more than 48 domains.
Data exchange support	Supports PLMXML, TCXML, and Multi-Site.	Supports JSON, OntoML, BMEcat.

There is very little difference between these two types of data in the user interface. These data types are, however, installed and configured differently. If you have questions, contact your classification administrator.

4. About the classification standard taxonomy architecture

The classification standard taxonomy (CST) is displayed through a hierarchy of groups containing master nodes that reference classes in which data is stored. This hierarchy is referred to as the *presentation layer*. The presentation layer is installed by default.

The presentation layer does not actually store the classes. It references classes that are stored in the CST layer. A node in the hierarchy can hold a reference to a class that stores data. Group nodes are used for organizational structure, and master nodes reference application classes in the CST layer. The CST layer also stores properties and key-LOVs. Classification data is stored in the Teamcenter server layer.



The significance of this architecture becomes apparent when **understanding how CST and traditional Classification (SML) work together**.

5. About the objects that you work with

IRDI

The International Registration Data Identifier (IRDI) originates from the ECLASS standard. This identifier is used by the classification standard taxonomy to uniquely identify all classes, properties, and key-LOVs. Most importantly, using an IRDI supports:

- Unique namespaces
- Revisioning

In CST, the IRDI is displayed using the following format:

*aaaa...a***#***bb*-*cccccc***#***nnn*

Example:

XMPL#01-CLS001#001

Where:

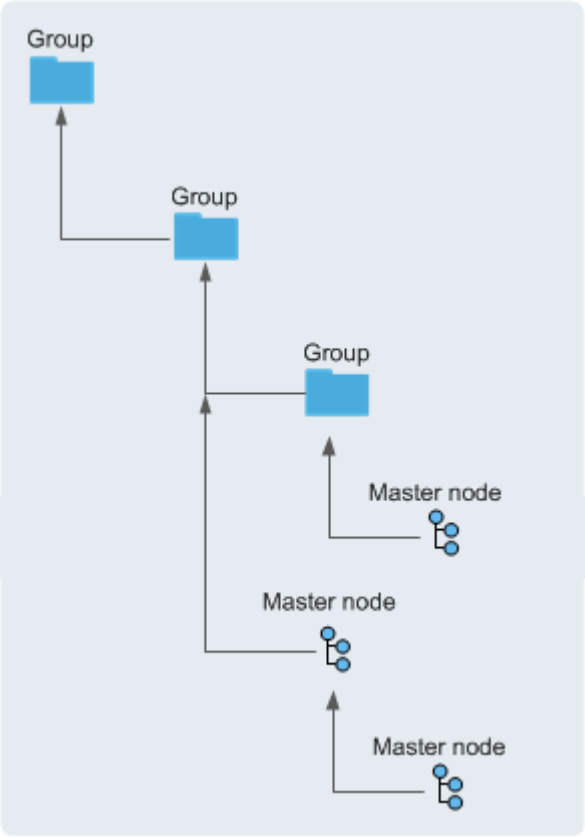
IRDI components	Description
aaaa...a	Unique alphanumeric namespace Numeric namespaces are reserved as companies and organizations can register them with ISO. For example, the following namespaces are reserved: **0173** is reserved by the ECLASS organization **0175** is reserved by Siemens Choose a namespace that is unique to your organization. This is particularly important if you plan to share data with other organizations.
#	Separator character
bb	Data type **01**: class **02**: property **09**: list of values (key-LOV)

IRDI components	Description
cccccc	Object identifier This identifier uniquely identifies the object within the object data type. This identifier can have up to six characters.
nnn	Revision number The revision number must have three characters.

Node hierarchy

The classification hierarchy consists of nodes of differing types.

- A *group* node is used for organization and cannot hold data.
- A *master* node references an application class that holds data. Master nodes can have other master nodes as children.



A node hierarchy can be revised.

Class

There are three types of classes in CST:

- Application class

  An *application class* is referenced by nodes and can be used to store data. They hold the properties used to define the class. An application class can reference properties and aspect classes.

- Property block class

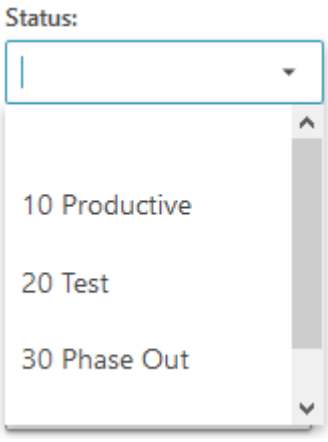
  A *property block class* groups sets of properties together. You can group properties that are frequently used, avoiding the need to repeatedly assign each property to a class. For example, a milling tool generally comprises a holder, an adapter, and an insert. Each of these components is described by many properties. The insert, for example, can be described by properties such as thickness, cutting length, grade, and material. Because these properties apply to every insert, you can group them into one property block class called **Insert** and reuse this property block like a simple property in every class that contains an insert.

- Aspect class

  An *aspect class* groups properties that pertain to overarching aspects of many objects and can be reused across multiple object types. They often group such parameters as commercial information, for example, supplier contact information, that have nothing to do with product definition. Aspect classes can only be referenced directly by application classes.

Key list of values

A *key list of values (key-LOV)* holds selection lists.



These lists can be nested.

Key-LOVs are referenced by properties.

Property

A *property* is used to describe attributes of a class. It can have multiple data types:

- String
- Integer
- Double
- Boolean
- Reference

Reference property types point to another object such as a key-LOV definition or a property block class.

Property group

A *property group* is the user interface representation of a property block class. It is a group of properties that you can navigate using the **Property Groups** pane.

Classification object

An internal classification object (ICO) is used to classify Teamcenter data.

View

Used to display all or a subset of the properties in a class in the user interface. There are two types of views used in classification classes:

- Base view
- User, group, or role view

6. ECLASS version support for the current version of Teamcenter

Advanced Classification in Teamcenter version 2412 supports all ECLASS versions from ECLASS 9 to ECLASS11.1.

7. Deploying classification

Planning the deployment of advanced classification

Before deploying classification, plan your deployment considering the following scenarios:

❒	You only want to use the advanced classification functionality.
❒	You want to use both basic and advanced classification functionality.
Additional scenarios are addressed when you have installed the advanced classification functionality:	
❒	You want to use the ECLASS taxonomy.
❒	You want to use AI assisted classification.

Once you have identified the scenarios that are applicable in your case, you can **install Teamcenter Classification** and then set up classification by creating the classification hierarchy and performing additional configurations.

Install classification using Deployment Center

Add the Classification application to your existing Teamcenter environment.

Procedure

1. Log on to Deployment Center and select the environment to which you want to add Classification.

2. Go to the **Applications** task. Click **Add or Remove Selected Applications** ✎.

3. In the **Available Applications** panel, use the web browser search to find the following applications depending on your scenario:

Scenario	Applications to select for Active Workspace based solution	Applications to select for Rich Client based solution
Advanced classification Based on new ISO, IEC standard classification system supporting advanced features like attribute blocks, cardinality, and polymorphism	**Advanced Classification** **Classification Active Workspace**	Not supported
Traditional basic classification and advanced classification together	**Advanced Classification** **Classification Active Workspace**	Not supported
Teamcenter Classification for ECLASS	**Advanced Classification**	Not supported

Scenario	Applications to select for Active Workspace based solution	Applications to select for Rich Client based solution
Importing of ECLASS Standard taxonomy into Teamcenter and classifying of Teamcenter data into ECLASS taxonomy	**Classification Active Workspace**	
AI assisted classification	**Classification Active Workspace**	Not supported
Classification localization support Supports internalization and localization of classification	**Classification L10N**	**Classification L10N**

Select the application, and then click **Update Selected Applications**.

Deployment Center automatically selects any additional dependent applications.

4. Go to the **Components** task.

5. In the **Selected Components** list, note any remaining components whose configuration status is not **100%**. Select each incomplete component, enter required parameters, and save component settings until all components in the environment show a configuration status of **100%**.

   When all components are fully configured, the **Deploy** task is enabled.

6. Go to the **Deploy** task. Click **Generate Install Scripts** to generate deployment scripts you will use to update affected machines.

   When script generation is complete, note any special instructions in the **Deploy Instructions** panel.

7. Locate deployment scripts, copy each script to its target machine, and then run each script on its target machine.

   For more information about running deployment scripts, see *Deployment Center — Usage*.

Postrequisites

If you want to use classification with Active Workspace, ensure that you **enable the presentation layer**.

Install classification using Teamcenter Environment Manager

You can install the classification functionality through Teamcenter Environment Manager.

Procedure

1. Launch Teamcenter Environment Manager.

2. Proceed to the **Features** panel and select the features based on your requirements.

Functionality required	Features to select
Install advanced classification	• **Extensions** > **Reuse and Standardization** > **Advanced Classification**
Use traditional basic classification and advanced classification with ECLASS data simultaneously	• **Active Workspace** > **Client** > **Reuse and Standardization** > **Classification Client** • **Active Workspace** > **Server Extensions** > **Reuse and Standardization** > **Classification Server** • **Active Workspace** > **Server Extensions** > **Reuse and Standardization** > **Presentation Layer - Next Generation Classification Server** • **Extensions** > **Reuse and Standardization** > **Advanced Classification**
Install AI based Classification	• **Microservices** > **Classification AI Serving** • **Active Workspace** > **Server Extensions** > **Reuse and Standardization** > **Classification AI**

3. After selecting the features perform the next steps to complete the installation.

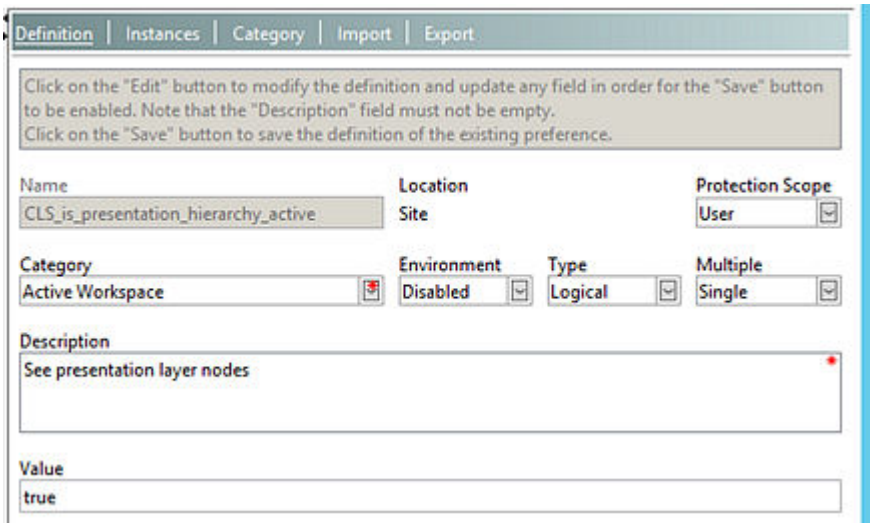
Configuring advanced classification

Enable the presentation layer

The presentation layer provides the ability to deal with traditional basic classification features (such as creating a class hierarchy, adding classification objects to a classification hierarchy, classifying workspace objects, searching for classification objects, and modifying and deleting objects from a classification hierarchy) and advanced CST and classification library management features simultaneously.

Additionally, the presentation layer adds the **Classification** tile to the **home page** and allows you to configure visual navigation cards for searching.

1. Make the node hierarchy (presentation layer) visible in the Active Workspace user interface by updating the value of the **CLS_is_presentation_hierarchy_active** user preference to **true**.



Being able to turn the visibility of the node hierarchy on and off per user allows users of both traditional classification and CST to classify objects in the same environment. Depending on the setting, a user can see either the traditional classification classes or the CST classes.

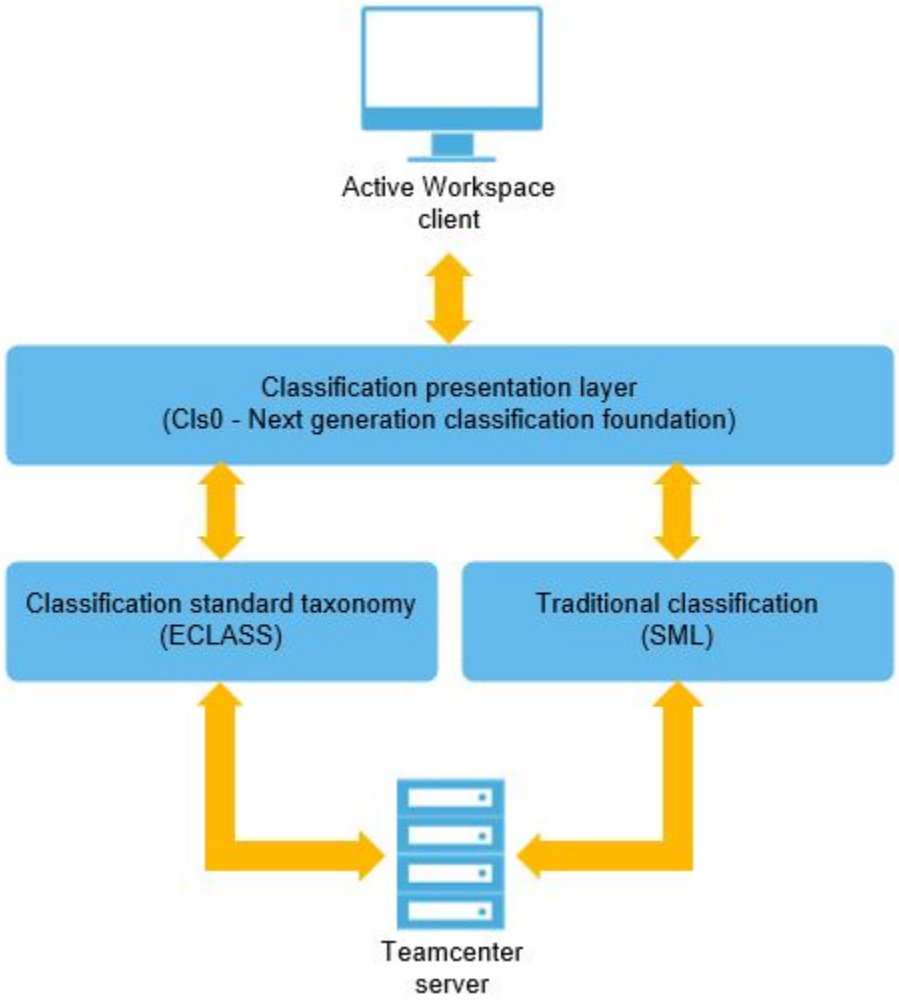
> Note:
>
> CST classes are not subject to access control. Therefore, if your environment contains both CST and traditional classification classes, if the presentation hierarchy is not active (**CLS_is_presentation_hierarchy_active=false**), CST classes are displayed in the search results for traditional classification users.

About working with advanced and basic classification simultaneously

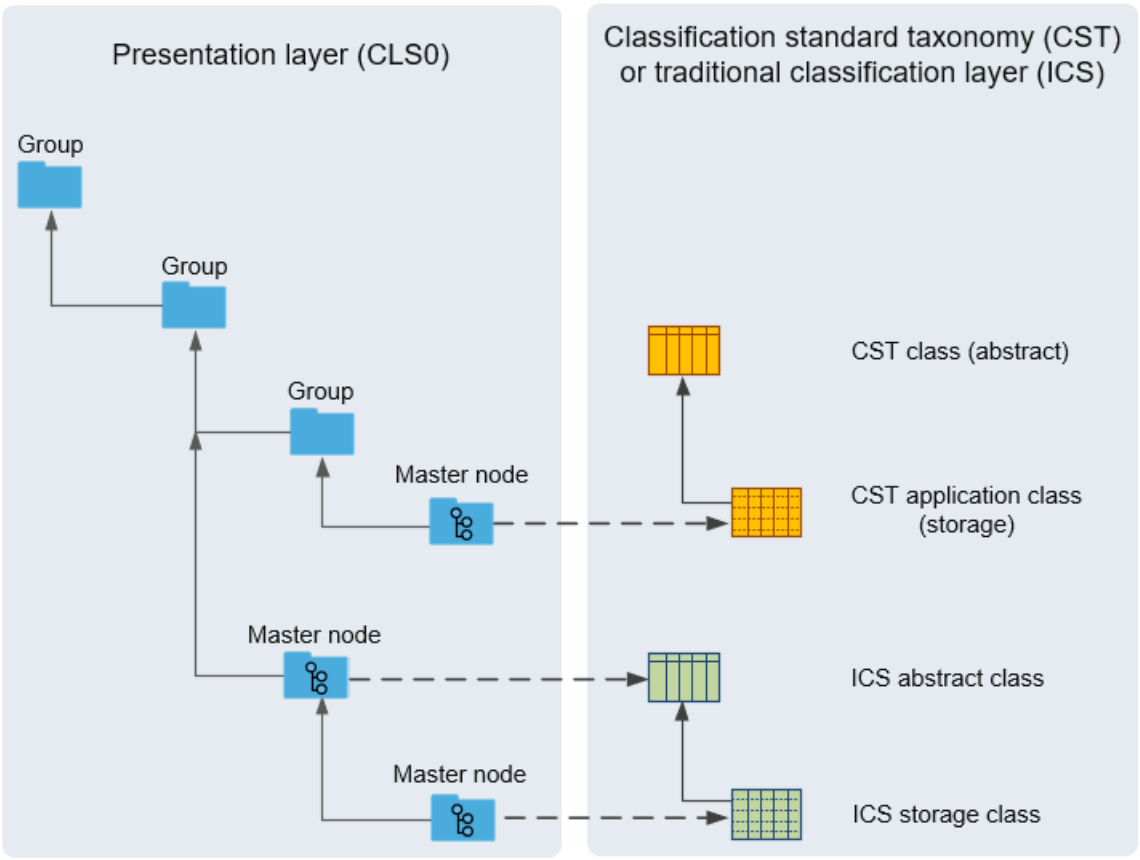
Classify with advanced and basic classification simultaneously

Advanced classification standard taxonomy (CST) offers some features that traditional basic classification does not. Additionally, in the future, CST will be regularly enhanced. For this reason, you may want to consider **migrating your traditional basic classification classes to the advanced CST format**.

Both advanced and basic classification classes can co-exist in a single environment.



In this case, the nodes reference the appropriate class type: For basic classification, a node references ICS classes and for advanced, it references CST classes.



The **CLS_is_presentation_hierarchy_active** user preference determines whether nodes or classes are displayed in the hierarchy. If this preference is set to **true**, only nodes are displayed in the hierarchy. If it is set to **false**, traditional classes are displayed.

> Note:
>
> The **CLS_is_presentation_hierarchy_active** preference is not delivered by default. You must add it manually.

Basic classification (ICS) classes must be extended to the presentation hierarchy as follows:

1. **Extend classes to the presentation layer using clsutility**.

2. **Synchronize the presentation layer with the classification hierarchy**.

Extend classes to the presentation layer using clsutility

1. Extend classification data to the presentation layer by running the following command line utility:

   clsutility -import -hierarchy -cid=*group-or-class-ID*

This command extends the classification subhierarchy under the specified group or class.

- For classification groups, running this utility creates presentation layer group nodes with the same ID as the associated groups.
- For classification classes, running this utility creates presentation layer master nodes with the **storage_class_type** property corresponding to their associated classes.

Additional optional arguments:

- **-include_instances**

  Includes ICOs in the subhierarchy.

- **-exclude_children**

  Excludes the subhierarchy. It extends the specified group or class only.

2. (Optional) **Synchronize the presentation layer with the classification hierarchy.**

Extend classes to the presentation layer when using only a subset of the classification hierarchy

When using only a subset of the classification hierarchy, you must run the **clsutility** on the entire hierarchy. In this case, the argument **-include_instances** is not required since you only need to extend the classes to the nodes that are required.

Example:

clsutility -import -hierarchy -cid=ICM

Synchronize the presentation layer with the classification hierarchy

Use one of the following methods to synchronize the presentation layer with the underlying classification hierarchy:

- **Manually run clsutility.**
- Enable automatic synchronization that shadows specific operations by setting the **CLS_auto_sync_node_hierarchy** preference to **true**.

  When at least the top-level node of the storage and presentation hierarchies are **linked**, the synchronization mechanism is triggered by a change to groups, classes, or ICOs in the storage hierarchy and results in the following changes to the presentation hierarchy.

Operation on class hierarchy	Operation required on node hierarchy
Add/delete group	Add/delete group node
Add/delete class	Add/delete master node
Add/delete class ICO	Add/delete classification element
Copy/paste or cut/paste of a group	Copy/paste or cut/paste of a group node
Copy/paste or cut/paste of a class	Copy/paste or cut/paste of a master node

The synchronization mechanism functions as follows:

- If a target is not found in the presentation hierarchy, it is created (applies to node or element).
  - To create a node in the presentation hierarchy, Teamcenter matches the parent node with the parent class.
  - To create an element in the presentation node, Teamcenter matches the node with the storage class.
- To find targets, Teamcenter searches for the following:
  - A group node with the same ID as the classification group
  - A master node with the storage class type property pointing to the associated classification class
  - A classification element referencing the associated ICO

  The objects in the hierarchies are mapped as follows.

Object types from Classification (class hierarchy)	Corresponding object types from the presentation layer (node hierarchy)
Group	Group node
Abstract class	Master node
Storage class	Master node
Classification object (ICO)	Classification element (ICO)

Migrating traditional basic classification to advanced classification standard taxonomy

Advanced classification standard taxonomy (CST) offers some features that traditional basic classification does not. Additionally, in the future, CST will be regularly enhanced. For this reason, you may want to consider migrating your traditional basic classification classes to the advanced CST format. To do this, you can run the following utility:

clsutility -migrate -classification2cst -cid=*class-ID* **[-recursive | -confirm| -include_icos]**

> Caution:
>
> It is strongly recommended that you test the migration of your classes in a staging environment and evaluate the results. Deploy the migration in production only when you are completely satisfied with the test results.

This utility migrates traditional classification classes and all the objects (properties, dictionary attributes, key-LOVs, and associated images) referenced by that class, as well as all the parent classes, to the CST format. Optionally, the utility can also convert all child classes (**-recursive**), and any ICOs already existing in the class (**-include_icos**).

Running the utility creates a class in the database with an automatically generated IRDI. This IRDI is displayed in the utility output. Additionally, node definitions are created for application classes so that the migrated CST classes are visible and can be used for classification.

```
Migration summary for traditional objects
-------------------------------------------------------------------------------------------------------------
               TYPE |          SML ID |                      CST ID |       Status | Note
-------------------------------------------------------------------------------------------------------------
     NodeDefinition |              AD |                 SMPL1#AD#001 |  UNPROCESSED |
 PropertyDefinition |          -80001 |           SMPL1#02-f6f89w#001 |  UNPROCESSED |
 PropertyDefinition |          -80002 |           SMPL1#02-fsf89w#001 |  UNPROCESSED |
 PropertyDefinition |          -80005 |           SMPL1#02-e6f89w#001 |  UNPROCESSED |
 PropertyDefinition |          -80015 |           SMPL1#02-ccf89w#001 |  UNPROCESSED |
 PropertyDefinition |          -80018 |           SMPL1#02-bsf89w#001 |  UNPROCESSED |
   KeyLOVDefinition |          -80051 |           SMPL1#09-Tcf89w#001 |  UNPROCESSED |
 PropertyDefinition |          -80020 |           SMPL1#02-bMf89w#001 |  UNPROCESSED |
    ClassDefinition |  ADT-CONNECTION |  SMPL1#01-ADT-CONNECTION#001 |  UNPROCESSED |
     NodeDefinition |  ADT-CONNECTION |     SMPL1#ADT-CONNECTION#001 |  UNPROCESSED |
 PropertyDefinition |          -82005 |           SMPL1#02-q5989w#001 |  UNPROCESSED |
   KeyLOVDefinition |          -80040 |           SMPL1#09-WMf89w#001 |  UNPROCESSED |
 PropertyDefinition |          -86123 |           SMPL1#02-la989w#001 |  UNPROCESSED |
    ClassDefinition |   ADT-CONN-BUND |   SMPL1#01-ADT-CONN-BUND#001 |  UNPROCESSED |
     NodeDefinition |   ADT-CONN-BUND |      SMPL1#ADT-CONN-BUND#001 |  UNPROCESSED |

JSON input for KeyLOVDefinitions: C:\Users\ny6wyo\AppData\Local\Temp\KeyLOVDefinitions.json
JSON input for PropertyDefinitions: C:\Users\ny6wyo\AppData\Local\Temp\PropertyDefinitions.json
JSON input for ClassDefinitions: C:\Users\ny6wyo\AppData\Local\Temp\ClassDefinitions.json
JSON input for NodeDefinitions: C:\Users\ny6wyo\AppData\Local\Temp\NodeDefinitions.json

Operation completed successfully.
```

The **-confirm** argument creates the objects in the database. It is recommended that you run the utility without the **-confirm** argument and review the resulting JSON files. These are created in your *temp* directory. The exact path is included in the utility output. You can modify the JSON files if necessary and import them manually or, if you are satisfied with them, rerun the migration utility using the **-confirm** argument.

You must set the following two preferences to run the migration:

- **CST_default_namespace**

  This preference specifies the namespace of all CST objects created by the migration utility.

- **CST_default_migration_status**

  This preference specifies the status of the CST objects created by the migration utility. Valid values are **Develop**, **Released**, or **Retired**. Objects set to **Released** can no longer be modified.

The output is displayed in tabular form with a column for status. This status can be one of:

- **UNPROCESSED**

  The migration utility was run in dryrun mode (default). You can review the JSON files created for the new object but it does not yet exist in the database.

- **SKIPPED**

  The reason that the object was skipped is displayed in the **Note** column.

- **SUCCESS**

  The migration utility was run with the **-confirm** argument. The object was successfully migrated to advanced classification.

- **FAILURE**

  The object was not successfully migrated. See the **Note** column for more information.

Limitations

- Class attribute, class detail, and view attribute options that are not supported in advanced classification are migrated as property values only. Your data is not lost, but is not relevant to advanced classification. Only the **Mandatory** and **Protected** options are supported in advanced classification.

- As advanced classification does not support localization for the **UserData1**, **UserData2**, and **Annotation** properties, localized values for these properties are not migrated.

- Default or minimum and maximum values set for properties are stored as individual property values. Advanced classification does not support inheritance, so if a default or minimum and maximum value is inherited in rich client, the value that would be displayed through that inheritance mechanism is displayed as the property value in the migrated class or view. If one of the supported views contains a value in rich client that differs from the value displayed in the default view, the property is assigned the value displayed in the default view.

- Reference attributes are not supported in CST.

- Interdependent key-LOVs are migrated as individual property values.

- Only views that are supported in advanced classification (**Default**, **User**, **Group**, and **Role** views) can be migrated.

Configuring search for classification

Configure search for advanced classification

After installing classification or after adding data (classes, views, properties), you must enable searching for the data by **indexing classification data**.

Any time you create new classification data, such as classifying new objects, you must re-index to be able to search for the new object in the global search or see the property displayed in the list of filters when searching for data. There are two ways to re-index:

- Index all data.

  This can be a time-consuming process depending on the amount of data in your database.

- **Index only the workspace objects whose classification data has changed**.

Make properties searchable in the global search

By default when you add properties to a class, they are displayed as searchable filters. If, additionally, to search for properties in the global search, or when indexing classification data to mark the changed classification objects, you must do the following:

1. Run the **bmide_modeltool** utility.

   For example, in a Teamcenter command window, change to the *TC_ROOT***bin** directory and type:

   ```
   bmide_modeltool -u=username -p=password -g=dba -tool=all
   -mode=upgrade
   -target_dir="TC_DATA"
   ```

2. Verify the schema file was correctly updated.

a. Verify that the **tc_solr_schema.xml** file in the *TC_DATA***ftsi\solr_schema_files** folder has the current date and time.

b. Open the **tc_solr_schema.xml** file in a text editor and search for **TC_0Y0_CLS_0Y0**. You should find several lines containing this string. Close the text editor without saving.

Searching using localized classification values

Classification values are localized along with other objects using the same localization process.

The following features are fully supported for localization:

- Keyword search
- Property specific search
- Category name in filter panel
- Filter value
- Filtering using localized value

Re-index classification data after upgrading Teamcenter

After upgrading Teamcenter, or after installing any classification features, you must **re-index classification data**.

If the new release to which you are upgrading includes new classification features, it generally also includes schema changes. During this upgrade procedure all classification data is marked for re-indexing and included during your next indexing synchronization. This can cause the synchronization to take a longer than usual.

Classification business constants for search

Business object constant	Object type on which constant can be set	Default setting	Description
Awp0SearchIsIndexed	**Item** **ItemRevision** **Cls0ClassBase** **Lbr0LibraryElement**	Set to **true** on the **ItemRevision**	Enables a business object for indexing. This business object is configured in the **AWS2** template owned by the search. Although it is possible to set this constant to true on **Item** objects, it is recommended to index item revisions only. To enable classification presentation layer classification objects (**Cls0ClassBase**) and library elements (**Lbs0LibraryElement**) for indexing, you must install the **Cls1ClassificationAW** and **Lbr0LibraryManagementAW** templates which

Business object constant	Object type on which constant can be set	Default setting	Description
			set this constant to **true** on **Cls0ClassBase** and **Lbr0LibraryElement** respectively.
Awp0SearchIsClassifyDataIndexed	Specific object type, for example: **Item** **ItemRevision** **Cls0Object** **Lbr0LibraryElement**	Is set automatically to **true** when you install the **ics1icsaw** template.	This business object type constant is specific to classification and is set on a business object to enable its associated classification information for indexing. If in an environment item revisions are classified and are also marked for indexing and if this business object type constant value is set to **true**, classification information associated to those item revisions are indexed. Administrators needs to install the **ics1icsaw** template to enable classification indexing which sets this BO constant to **true** on **workspaceObject**. All subtypes of **WorkspaceObject** (**Item**, **ItemRevision**, and so on) inherit this configuration.
Awp0SearchClassifySearchEnabled	**WorkspaceObject**	Installing the **ics1icsaw** template to enable classification data indexing, sets this business object type constant to **true** on **WorkspaceObject** and all its subtypes inherit this configuration.	This business object type constant is set to enable indexed classification properties for search. If this business object type constant is set to **false** and if the classification information is already indexed, it is not used for searching. Consequently, classification-specific filter categories are not displayed in the Active Workspace client.

Configuring the classification interface

Adding images and icons to a classification class or hierarchy

A visual representation of classes helps to quickly navigate the hierarchy or provides additional information about a class. In classification, you can add images and icons to a class.

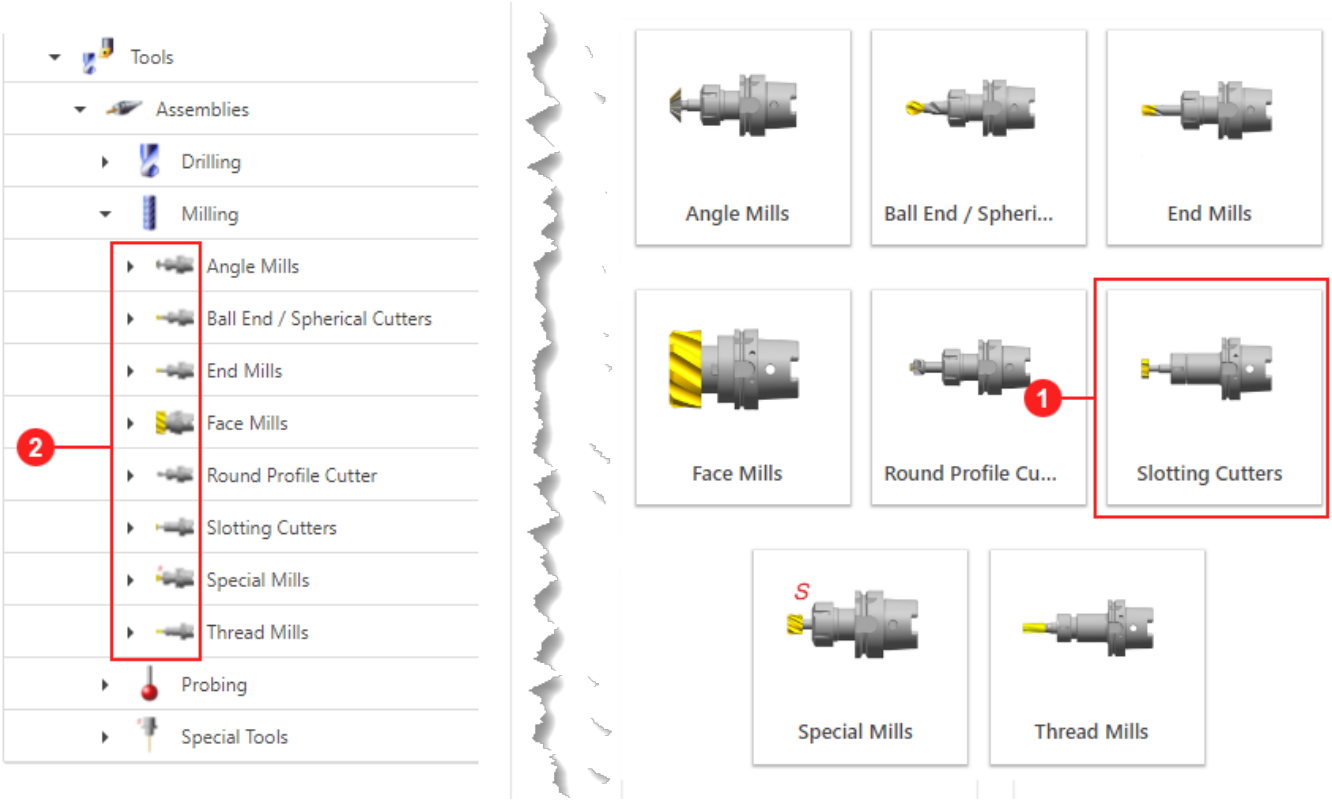
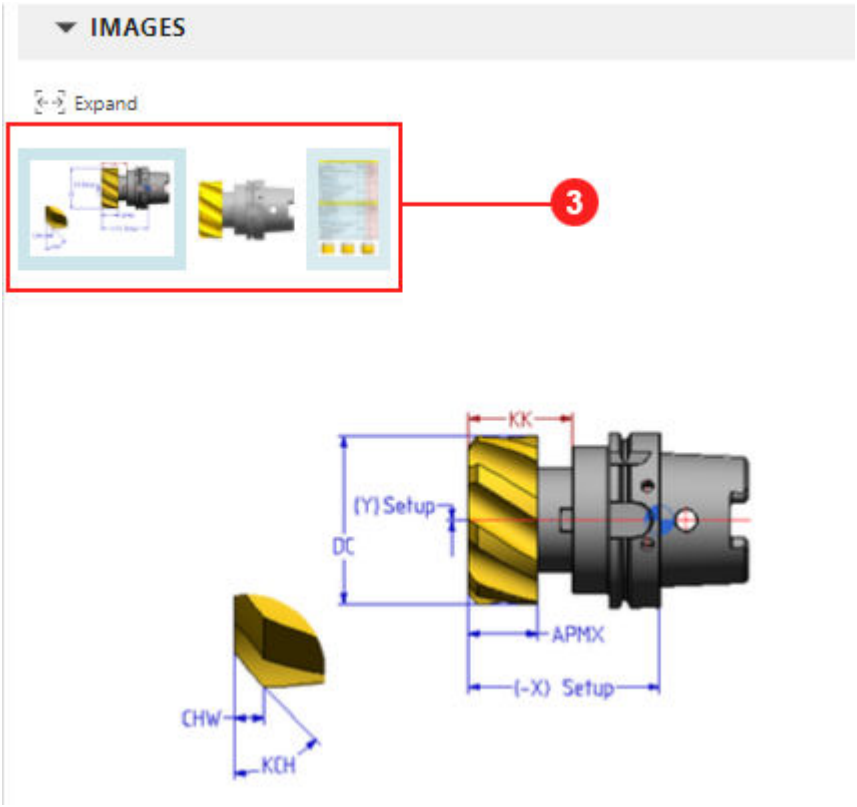


Image 1 Refers to an image associated with a class. A class can have multiple images associated. These are displayed in the **IMAGES** pane where you can switch between them using the carousel 3.



One image is designated as the primary image. This is the image that is displayed in the visual navigation card.

Icon 2 Refers to the small graphic displayed in the classification hierarchy when you navigate the hierarchy.

The behavior of images and icons varies depending on what type of classification you use.

Import icons and images using the following **clsutility** commands:

-import -image -id=*hierarchy-node-ID* **-os_path=***path-to-image-file*

-import -icon -id=*hierarchy-node-ID* **-os_path=***path-to-icon-file*

Add images and an icon to a node

Adding images or an icon to a node provides visual points of reference within the classification hierarchy. Images help to identify the contents of a node. You can add multiple images to a node which then appear in the **Preview** section within the **Overview**.

Procedure

1. On the **Classification Manager** dashboard, click **Nodes**.

2. Select a node to which you want to add an image or an icon and then select **Attachments**.

3. In the **Files** section, click **Add** ⊕ and click any one of the following:

   - **Add Icon** ⊕ to add an icon in the selected node.
   - **Add Image** ⊕ to add an image in the selected node.

4. In the **Add Icon** or **Add Image** panel, click **Choose File** to select the required file, and click **Add** to add an icon or an image to the node.

   When you add an image as an attachment to a node, a preview of the image appears in the **Preview** section of the **Overview**. For a given node, you can include multiple images by adding them one at a time.

   When you add an icon as an attachment to a node, it is displayed in the visual navigation card of the node tree. You can add one icon at a time. To replace an icon, you must first delete the previously added icon.

5. Click **List** or **Images** to view all the images and icon added to the node.

Delete an image or an icon from a node

You can delete an image or icon from a node.

Procedure

1. On the **Classification Manager** dashboard, click **Nodes**.

2. Select a node to from which you want to delete the image or icon.

3. Under **Files**, select the image or icon and click **Delete** ✕.

4. In the confirmation message, click **Delete**.

Changing the default icon of visual navigation cards

In the Class Navigator you can view the classification hierarchy by image. These tile images are referred to as *visual navigation cards* (VNCs) and allow the display of larger images while viewing the selected level in the classification node hierarchy.

VNCs are visible in the following areas:

- The Classification location when you click on the CLASSIFICATION tile on the home page

- The **Classification** tab where you classify objects

Configuring the image displayed on the VNC

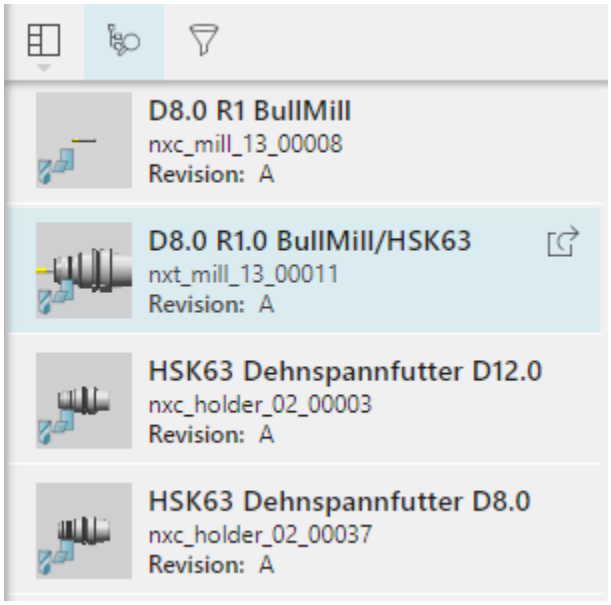
If you have not yet imported node icons, a generic VNC is displayed listing the children of the node.

The images are based on the node icons that you import using **clsutility -import -icon -id=***hierarchy-node-ID*. If a node icon does not exist for a class, there is a fallback mechanism that searches for the first image of a node, if none exists, then the first class image is used.

> Tip:
>
> To make optimal use of the image space on the VNC, create a square image. Rectangular images are cropped.

Configuring what information appears in the classification tiles

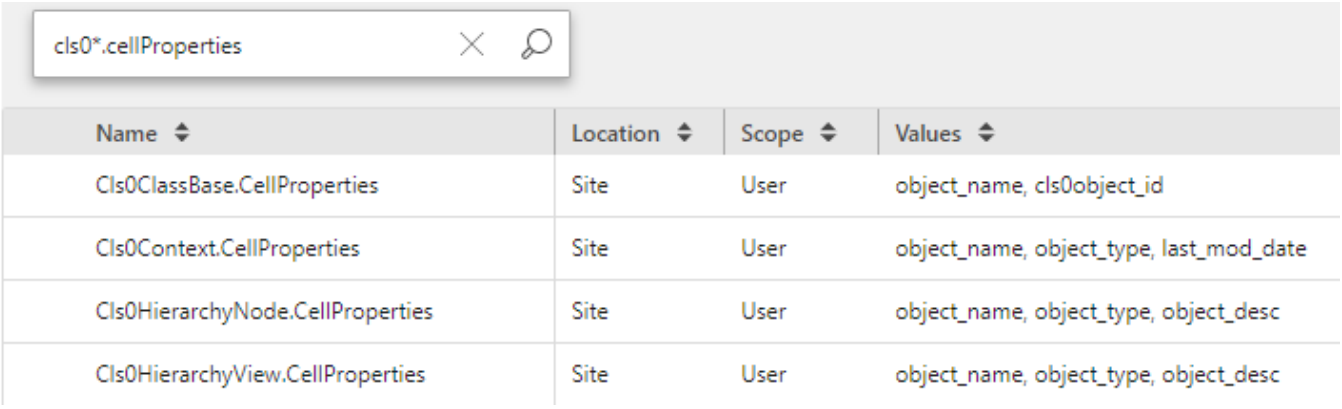
Tiles are displayed in the primary work area, for example, in search results:



Configuration of the object tiles respects the type hierarchy.

Mdl0ModelElement		
	Cls0ClassBase	
		Cls0CstObject
		Cls0Object

Tile properties are configurable on the parent class or on the child classes themselves. Site preferences determine what is displayed on the tiles for **Cls0Object** and **Cls0CstObject**, for example:



Name	Location	Scope	Values
Cls0ClassBase.CellProperties	Site	User	object_name, cls0object_id
Cls0Context.CellProperties	Site	User	object_name, object_type, last_mod_date
Cls0HierarchyNode.CellProperties	Site	User	object_name, object_type, object_desc
Cls0HierarchyView.CellProperties	Site	User	object_name, object_type, object_desc

Enable the display of DWG and CGM class images in the Classification tab

To use CGM or DWG file formats as class graphics, you must first convert them to PDF using the **prepare** utility delivered with the Lifecycle Visualization installation.

1. Run the Lifecycle Visualization installation, and select the **Convert and Print** option.

2. Copy the CGM or DWG file to the **VVCP** directory in the installation location.

3. Double click the **prepare.exe** utility.

4. Run the utility to convert the **CGM** or **DWG** files to PDF. For example, for a file named **my_drawing.dwg**, enter the following:

   ```
   prepare -pdf my_drawing.dwg
   ```

   This converts the **my_drawing.dwg** file and creates the **my_drawing.pdf** file in the **VVCP** directory.

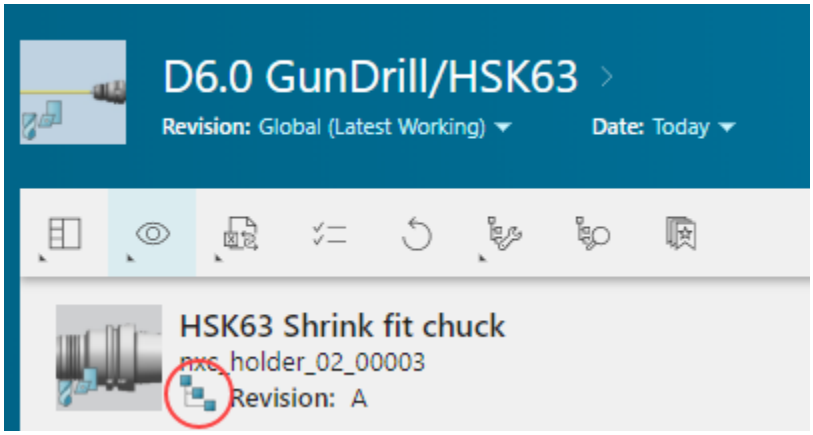
   You can add path names to change the input and output directories.

   Run `prepare -h` to obtain information about the utility.

5. Associate the **PDF** to the required class in Classification Admin (rich client).

Configure a visual indicator for classification on an object

You can set a visual indicator on an object to display that it is classified:



Add the following to your configuration code:

```
"classiBOM": {
            "iconName": "Classified",
            "tooltip": {
                "showPropDisplayName": false,
                "propNames": ["ics_classified"]
            },
            "modelTypes": ["Awb0DesignElement"],
            "prop": {
                "names": ["awb0UnderlyingObject"],
                "type": {
                    "names": ["ItemRevision"],
                    "prop": {
                        "names": ["ics_classified"],
                        "conditions": {
                            "ics_classified": {
                                "$eq": "YES"
                            }
                        }
                    }
                }
            }
        }
```

Setup advanced classification with ECLASS data

ECLASS and Teamcenter terminology

The classification standard taxonomy objects and concepts have different nomenclature than the ECLASS objects.

The ECLASS website provides in-depth explanations of ECLASS objects:

https://eclass.eu

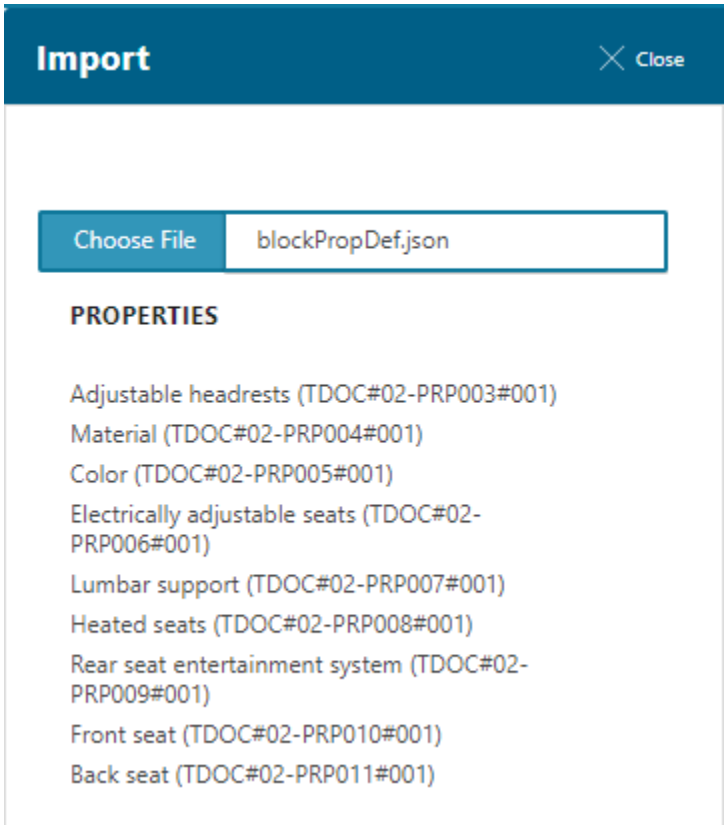
The most commonly used objects are as follows.

ECLASS	CST	Traditional classification (ICS)
Value list	Key-LOV definition	Key-LOV
Property	Property definition	Dictionary attribute
Classification class	Node definition	--
Application class	Class definition	Group/class
IRDI	IRDI	ID
Property block	Property block	--
Application data	Classification object	ICO
Aspect	Aspect	--

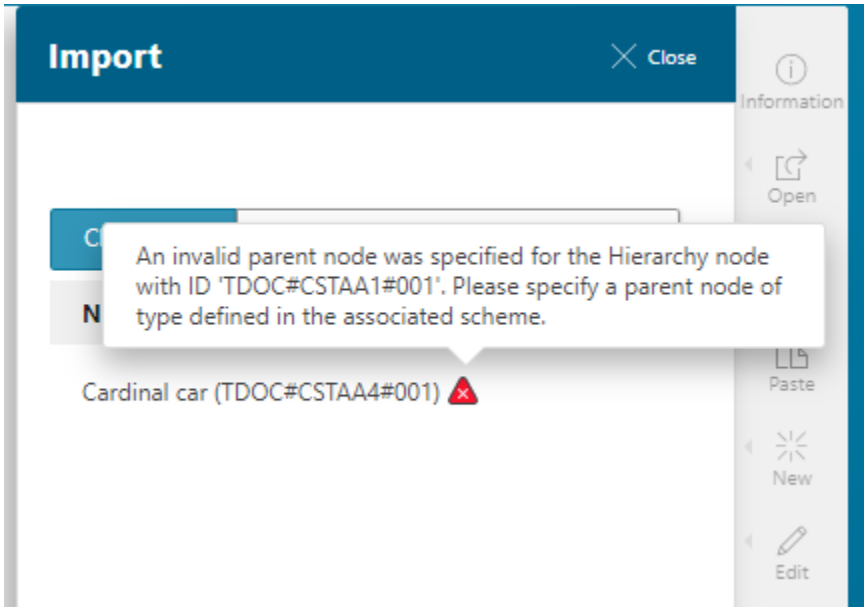
Importing and exporting classification data

Importing hierarchy definitions with the Classification Manager

For definition files based on schemas prior to version 1.5, you can import the definitions directly in the Teamcenter user interface. Use the **Import** button on the **Dashboard** tab or the **Import** command on any of the other tabs in **Classification Manager** to import key-LOV, property, class, and node definitions. They must always be **imported in the correct order**. After you upload the JSON file, you receive an overview of the objects you are about to import.



If there is an error in the files that you import, the error message provides assistance.



Tip:

It does not matter on which tab you choose to import, you can import any type of hierarchy definition on any of the tabs.

Import classified data for advanced classification

You can import classified data in BMEcat XML or JSON format in the user interface. The XML files and supporting documents (for example, images) must be available in a local directory that you compress to a ZIP file. The import searches the compressed for the correct input file.

1. Compress the directory structure of the data in a ZIP file.

   If you download BMEcat files from the Siemens Industry Mall, these are already in the appropriate zipped format.

2. In your **Home** folder, choose **More Commands** ••• > **Import/Export** > **Import Classification Data**.

3. In the **Choose File** box, select the ZIP file containing the data.

   Teamcenter searches inside the compressed file for all eligible import files and presents them to you in a list.

4. In the **Select file to import** box, select the XML file containing the data.

   If you import BMEcat data, there may be several versions of the XML file in different languages. These are not localized values. If you import, for example, the English version, and then subsequently import the German version, the German version overwrites the English values. Each of these language files represent master locales.

When the import is complete, you are notified in the **Alert** area where you can view the details of the import. As soon as the data is indexed, it is available for searching.

This feature requires that both the Subscription Manager and Dispatcher Client are running. If the Subscription Manager is not running, no notification is displayed in the **Alert** area, but the operation still occurs in the background.

If the imported data includes classification objects only, at import they are automatically attached to existing objects in the database if they have the same ID. This automatically classifies the existing objects or updates an existing classification on previously classified objects. If there is no existing object with the same ID in the database, a new object is created.

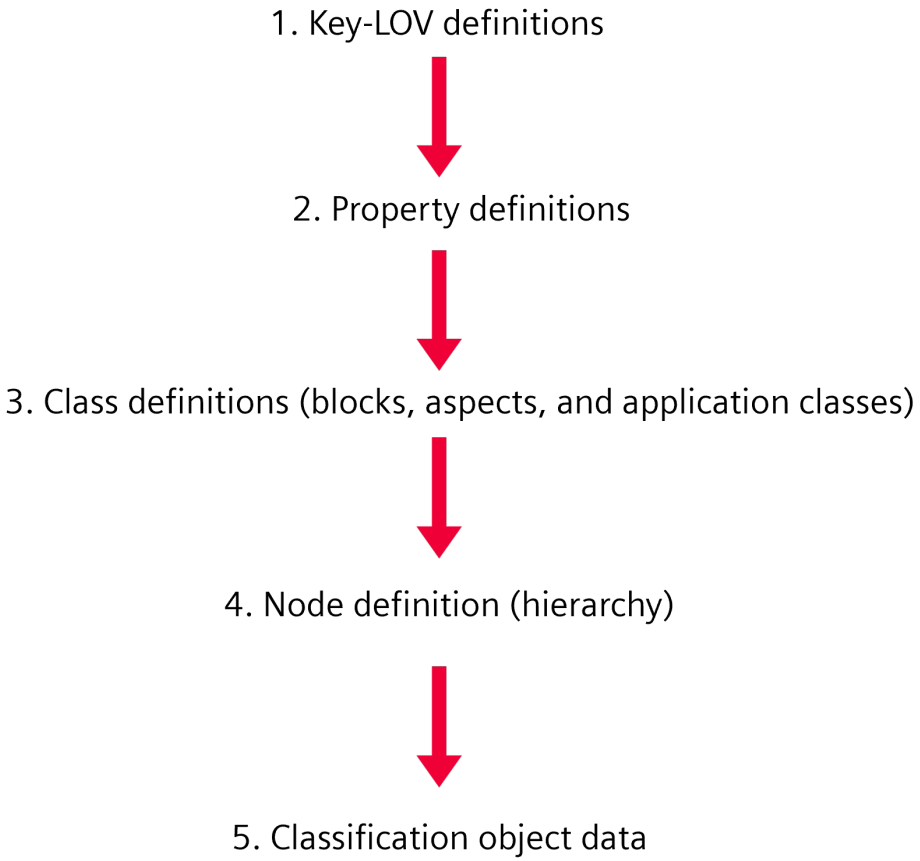
Importing JSON files using the clsutility command

You create a CST hierarchy by importing JSON files containing both hierarchy and data. The schema files defining the JSON format and example files are in the following directory:

...*TC_DATA*\classification\json\schema\advanced

The schema used to import classification definitions is evolving over time. Prior to version Teamcenter 14.1 (schema 1.5.0), key-LOVs, properties, class, and nodes had to be defined in separate JSON files and imported in the order listed. Using schema 1.5.0, you can import all types of class definitions in a single JSON file. This schema also supports importing older object definition JSON files with no need to rewrite them. You can define new objects or update existing ones in the same JSON file.

When importing, you can only reference objects that are already in the database. When importing classes that reference other classes (for example, when importing aspects), you must import the referenced classes first and then import the referencing classes. The definition file is read from top to bottom. If you import individual files, the order of import is as follows:



Data is imported from JSON files using the following methods:

- **Use the Classification Manager to import JSON files** in the user interface.

- Use **clsutility** to import a definition file based on any schema up to and including version 1.5.

  clsutility -u=*user* **-p=***password* **-g=***group* **-save -classification_definitions -request="***path-to-JSON-file***"**

- Use **clsutility** to import object individual definition files.

  The syntax of this utility is:

  clsutility-u=*user* **-p=***password* **-g=***group -operation input/output file path*

  If using schema 1.5, import the definition file using the following syntax:

  clsutility -u=*user* **-p=***password* **-g=***group* **-create -keylov_definitions -request="***path-to-JSON-file***"**

  For more information about the available commands and their exact syntax, type **clsutility -h** for the top-level help and **clsutility -***command-name* **-h** for information about each available command.

  The following table lists the supported objects and an example of the **clsutility** statements used to import these object types.

Object	Operation	clsutility command
Key-LOV	Import	**clsutility -u=***user* **-p=***password* **-g=***group* **-create -keylov_definitions -request="***path-to-JSON-file***"**
Property definition	Import	**clsutility -u=***user* **-p=***password* **-g=***group* **-create -property_definitions -request="***path-to-JSON-file***"**
Class definition	Import	**clsutility -u=***user* **-p=***password* **-g=***group* **-create -class_definitions -request="***path-to-JSON-file***"**
Nodes	Import	**clsutility -u=***user* **-p=***password* **-g=***group* **-create -node_definitions -request="***path-to-JSON-file***"**
Application data (ICOs)	Import	**clsutility -u=***user* **-p=***password* **-g=***group* **-create -classification_objects -request="***path-to-JSON-file***"**

Note:

There are pairs of very similar commands in the **clsutility**: **classification_objects** and **classification_object**, or **node_definitions** and **node_definition**. Be sure to use the plural forms of these commands (**objects** and **definitions**) for all CST imports. The singular refers to traditional classification objects.

Preparing JSON files for import (example)

Importing the hierarchy and classes of car seats

This example walks you through creating and importing the JSON files required to describe a class containing attributes that allow you to configure the seats in a car. The example consists of several levels of complexity:

- Classify a car in a class containing simple properties, for example, a key-LOV with the type of seat (front seat or back seat).
- Classify a car in a class containing reusable sets of properties (blocks).
- Specify the number of seats when classifying the car (cardinality).
- Specify the attributes shown depending on the type of seat selected (polymorphism).
- Specify the attributes shown for each seat when specifying multiple seats (cardinality and polymorphism).
- Import a property record into one of the newly created classes.

For each of these examples, the JSON files are explained. Once created, the JSON files can be imported using the Classification Manager or, alternatively, with **clsutility**. The **clsutility** commands required to import the JSON files are also listed **here**.

To verify that the definitions in the following examples are correct or to create additional definitions, consult the following schema files found in **...***TC_DATA***\classification\json\schema\advanced**:

- **Classification_Save_KeyLOVDefinitions_Request_advanced.schema.json**
- **Classification_Save_PropertyDefinitions_Request_advanced.schema.json**
- **Classification_Save_ClassDefinitions_Request_advanced.schema.json**
- **Classification_Save_NodeDefinitions_Request_advanced.schema.json**
- **Classification_Save_PropertyRecords_Request_advanced.schema.json**

There are many online JSON validators to assist you in creating JSON files with the proper syntax. We recommend using any of these to verify the JSON files that you create prior to importing them.

Note the following points about these examples:

- The definitions are imported with a release status of **Develop**. This ensures that you can still modify them if required. When you are satisfied with the structure and classes, you must **change the status to Released**. Objects can be classified only in **Released** classes.

- The **TDOC** name space is used to differentiate the structures imported in this example from other structures that you use. Creating your own name space helps when creating queries. When searching for your newly-imported objects, you can filter the tabs in Classification Manager to only show those with **Namespace** = **TDOC**.

Import a class with simple properties

The simplest use case is to import a class containing properties. This example shows how to import a **Car** class containing three properties:

- **Type of seat**

  A key-LOV with two entries: **Front** and **Back**

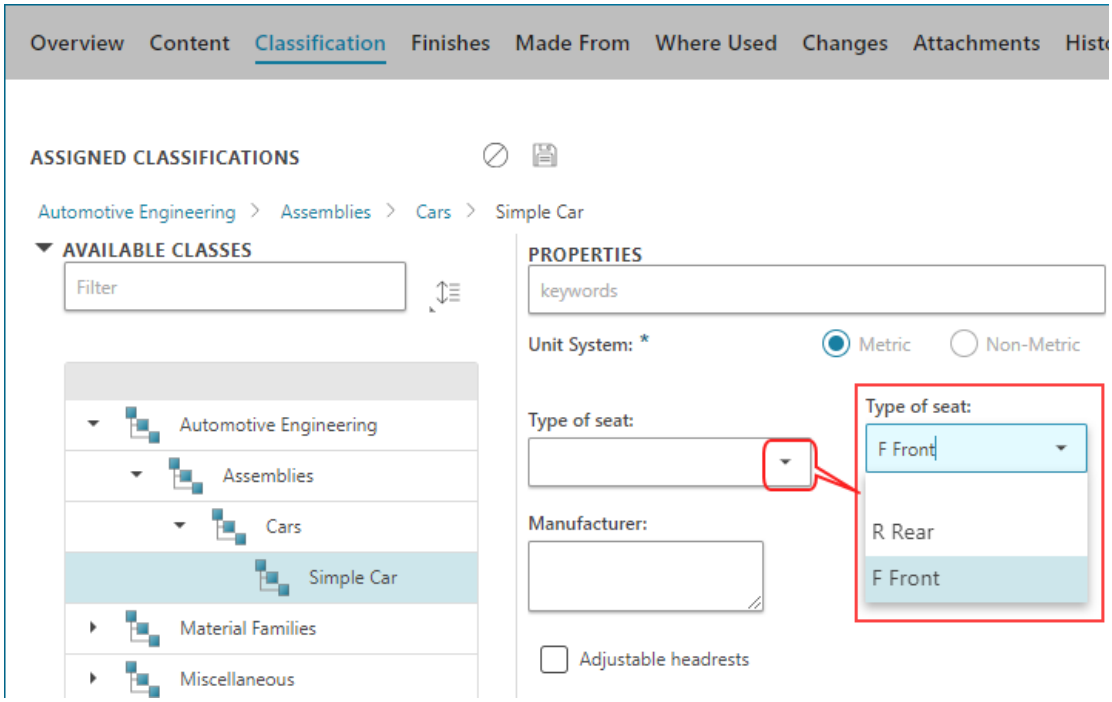
- **Manufacturer**

  A simple string property

- **Adjustable headrests**

  A Boolean property

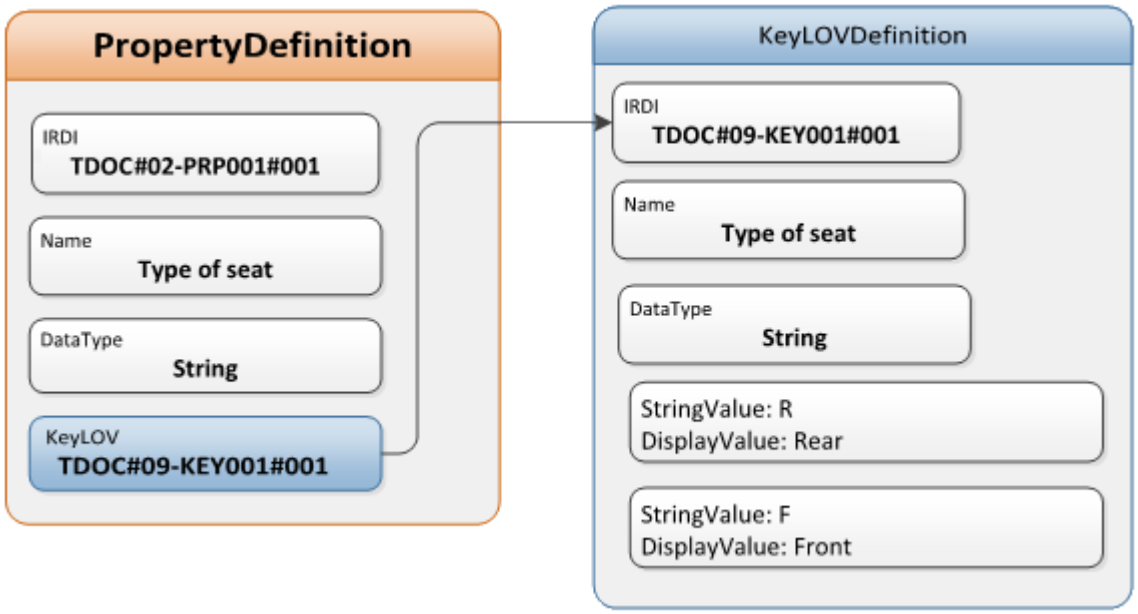
In the user interface, this class is displayed as follows:



The order when importing this class is to first import the key-LOV definitions, then the property definitions, then the class definition, and finally the node definition so that the class is visible in the classification hierarchy.

1. Import the key-LOV definition.

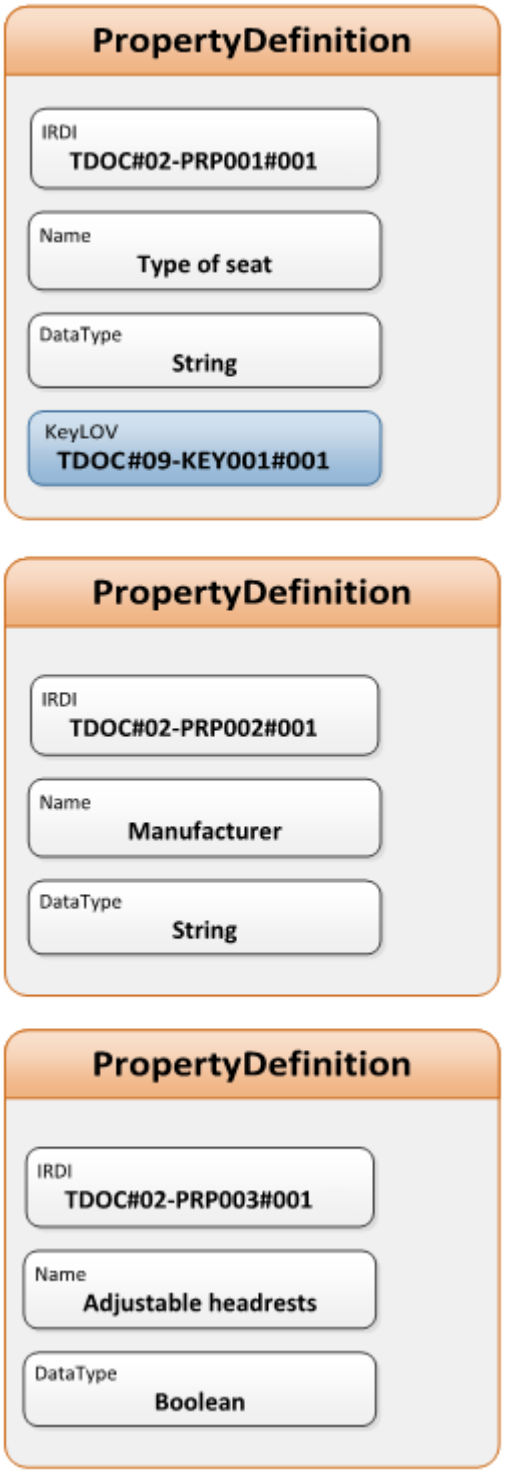
   In classification standard taxonomy, a key-LOV definition is separate from its property definition.



In this example, there is one key-LOV whose JSON definition is stored in a file titled **simpleKeylovDef.json**:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "KeyLOVDefinitions": [
    {
      "ObjectType": "09",
      "Namespace": "TDOC",
      "ID": "KEY001",
      "Revision": "001",
      "Name": "Type of seat",
      "Status": "Develop",
      "LOVItems": {
        "DataType": "String",
        "LOVStringItems": [
          {
            "StringValue": "R",
            "DisplayValue": "Rear Seat"
          },
          {
            "StringValue": "F",
            "DisplayValue": "Front Seat"
          }
        ]
      }
    }
  ]
}
```

2. Import the property definitions.



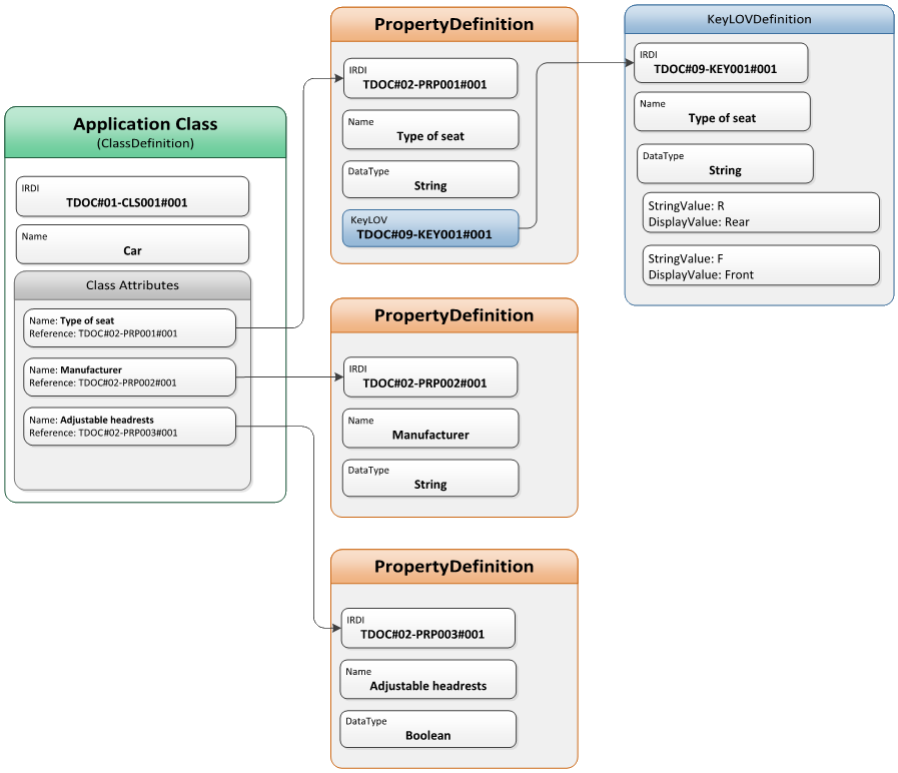
In this example, there are three properties to import, one of which (**Type of seat**) is a key-LOV that references the key-LOV definition imported in the previous step. The JSON definition for the properties is stored in a file titled **simplePropDef.json**:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "PropertyDefinitions": [
        {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP001",
      "Revision": "001",
      "Status": "Develop",
      "Name": "Type of seat",
      "DataType": {
        "Type": "String",
        "KeyLOV": "TDOC#09-KEY001#001"
      }
    },
     {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP002",
      "Revision": "001",
      "Name": "Manufacturer",
      "Status": "Develop",
      "DataType": {
        "Type": "String"
      }
    },
        {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP003",
      "Revision": "001",
      "Name": "Adjustable headrests",
      "Status": "Develop",
      "DataType": {
        "Type": "Boolean"
      }
    }


  ]
}
```

3. Import the class definition.

   The **Car** class contains only three properties: **Type of seat**, **Manufacturer**, and **Adjustable headrests**:



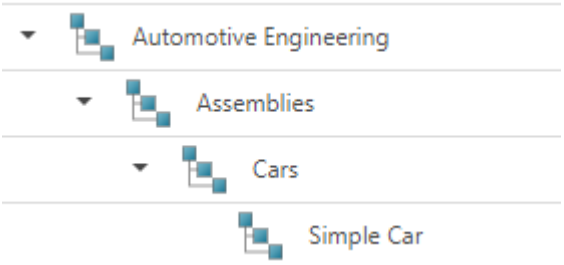
The JSON file, named **simpleApplicationClassDef.json**, consists of the following:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS001",
      "Revision": "001",
      "Name": "Car",
      "Status": "Develop",
      "IsAbstract": false,
      "UnitSystem": 3,
      "ClassType": "Application Class",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP001#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP002#001"
        },
        {
```

```
            "Type": "Property",
            "Reference": "TDOC#02-PRP003#001"
          }
        ]
      }
    ]
  }
```

4. Import the node definition that makes the class visible.

   A nested hierarchy is required to organize the classes.



   To import this hierarchy, create the following JSON file and name it **simpleNodeDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "NodeDefinitions": [
    {
      "Namespace": "TDOC",
      "ID": "CSTAE0",
      "Revision": "001",
      "Name": "Automotive Engineering"
    },
    {
      "Namespace": "TDOC",
      "ID": "CSTAA0",
      "Revision": "001",
      "Parent": {
        "Namespace": "TDOC",
        "ID": "CSTAE0",
        "Revision": "001"
      },
      "Name": "Assemblies"
    },
    {
      "Namespace": "TDOC",
      "ID": "CSTAA1",
      "Revision": "001",
```

```
        "Parent": {
          "Namespace": "TDOC",
          "ID": "CSTAA0",
          "Revision": "001"
        },
        "Name": "Cars"
      },
      {
        "Namespace": "TDOC",
        "ID": "CSTAA2",
        "Revision": "001",
        "Parent": {
          "Namespace": "TDOC",
          "ID": "CSTAA1",
          "Revision": "001"
        },
        "Name": "Simple Car",
        "ApplicationClass": {
          "Namespace": "TDOC",
          "ID": "CLS001",
          "Revision": "001"
        }
      }
    ]
  }
```

5. Import the definitions into the database using **Classification Manager** or with **clsutility commands**.

   You must import the JSON files in the following order:

   a. **simpleKeylovDef.json**

   b. **simplePropDef.json**

   c. **simpleApplicationClassDef.json**

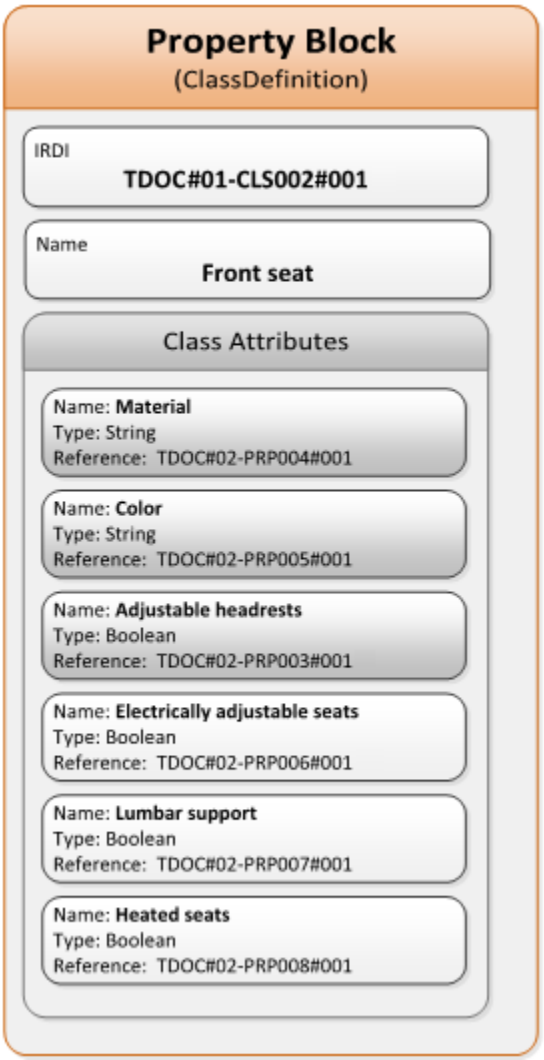
   d. **simpleNodeDef.json**

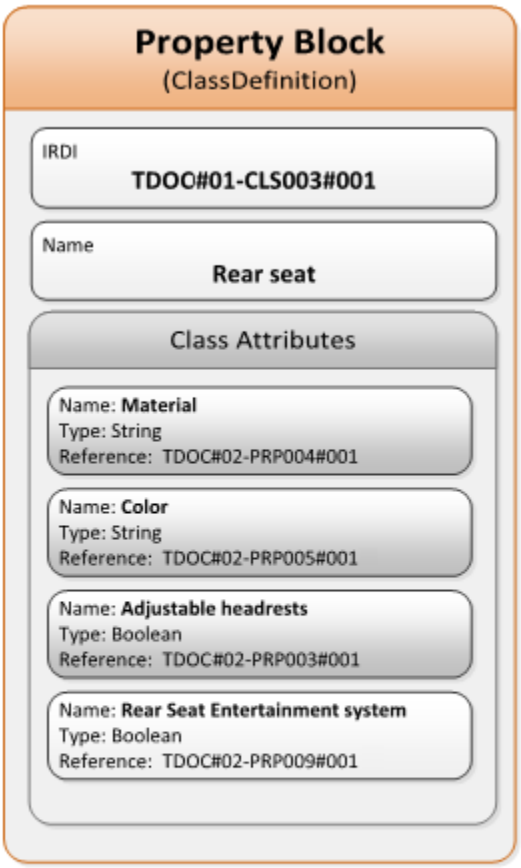
Import a class with a reusable set of properties (blocks)

Frequently, there is a group of properties that is always used together. For example, the following set of properties describe a front seat and a rear seat:

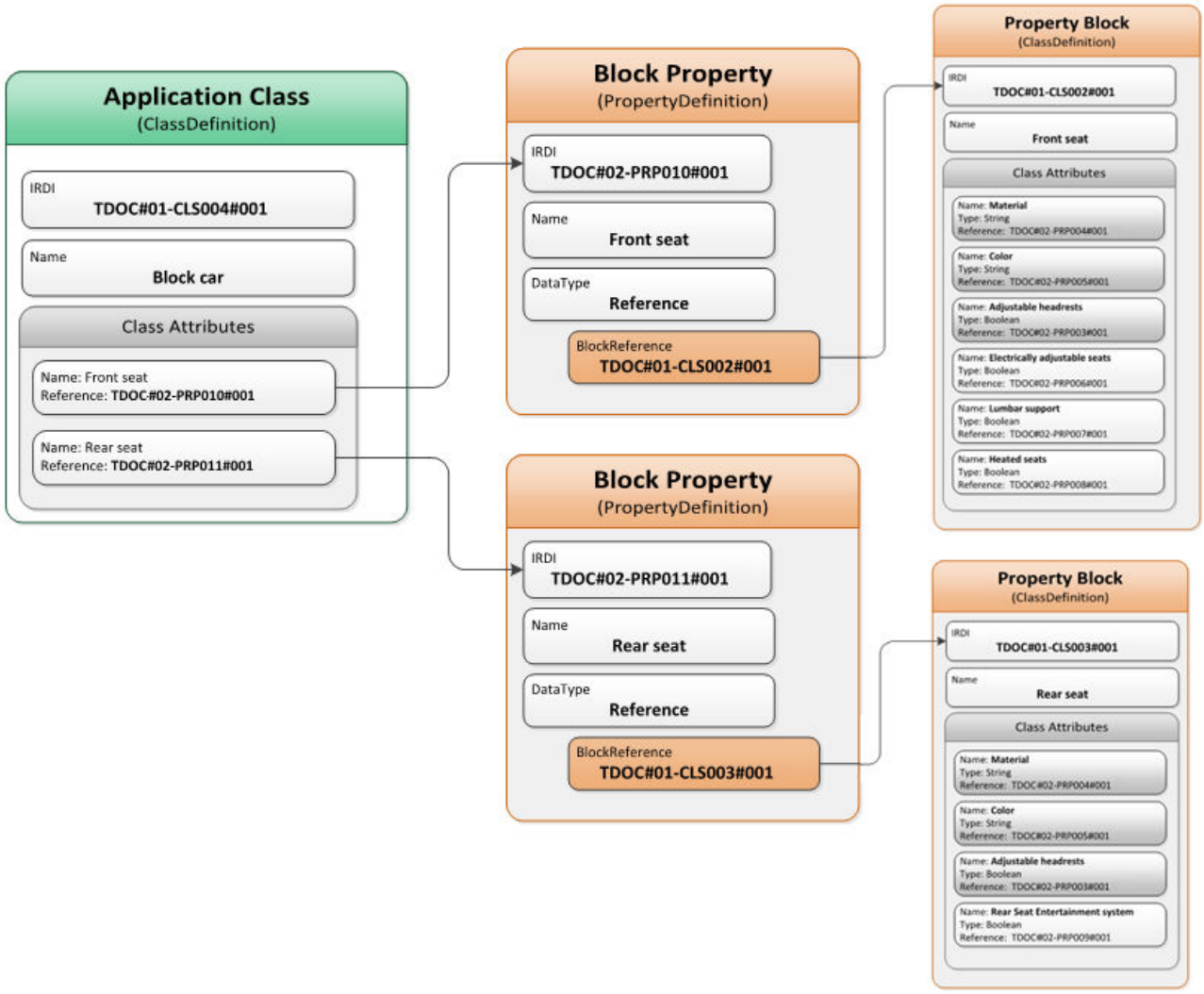
Front seat	Rear seat
Material	Material
Color	Color
Adjustable headrests	Adjustable headrests
Electrically adjustable seats	Rear seat entertainment
Lumbar support	
Heated seats	

Each time you create a class, you could manually add each of these properties to the class to describe each of these seat types. However, it is easier to group them and select a single property called **Front seat** that contains all the required properties. This set of properties is called a property block. A property block is a special kind of class.





Property blocks are referenced by block properties which, in turn, are referenced in the application class containing the block properties:



To create the JSON files required to import these classes and properties:

1. Create the individual property definitions.

   Create the following JSON file and save it in a file named **blockPropDef.json** as follows:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "PropertyDefinitions": [
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP004",
```

```
    "Revision": "001",
    "Status": "Develop",
    "Name": "Material",
    "DataType": {
      "Type": "String"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP005",
    "Revision": "001",
    "Name": "Color",
    "Status": "Develop",
    "DataType": {
      "Type": "String"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP003",
    "Revision": "001",
    "Name": "Adjustable headrests",
    "Status": "Develop",
    "DataType": {
      "Type": "Boolean"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP006",
    "Revision": "001",
    "Name": "Electrically adjustable seats",
    "Status": "Develop",
    "DataType": {
      "Type": "Boolean"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP007",
    "Revision": "001",
    "Name": "Lumbar support",
    "Status": "Develop",
    "DataType": {
      "Type": "Boolean"
```

```
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP008",
    "Revision": "001",
    "Name": "Heated seats",
    "Status": "Develop",
    "DataType": {
      "Type": "Boolean"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP009",
    "Revision": "001",
    "Name": "Rear seat entertainment system",
    "Status": "Develop",
    "DataType": {
      "Type": "Boolean"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP010",
    "Revision": "001",
    "Name": "Front seat",
    "Status": "Develop",
    "DataType": {
      "Type": "Reference",
      "BlockReference": "TDOC#01-CLS002#001"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP011",
    "Revision": "001",
    "Name": "Back seat",
    "Status": "Develop",
    "DataType": {
      "Type": "Reference",
      "BlockReference": "TDOC#01-CLS003#001"
    }
  }
```

```
    ]
  }
```

The last two properties, **PRP010** and **PRP011** reference the two property block classes that hold the sets of properties.

2. Create the class definitions, including the property block classes.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS002",
      "Revision": "001",
      "Name": "Front seat",
      "Status": "Develop",
      "IsAbstract": true,
      "UnitSystem": 3,
      "ClassType": "Block",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP004#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP005#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP003#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP006#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP007#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP008#001"
        }
      ]
    },
```

```
{
  "ObjectType": "01",
  "Namespace": "TDOC",
  "ID": "CLS003",
  "Revision": "001",
  "Name": "Back seat",
  "Status": "Develop",
  "IsAbstract": true,
  "UnitSystem": 3,
  "ClassType": "Block",
  "ClassAttributes": [
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP004#001"
    },
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP005#001"
    },
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP003#001"
    },
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP009#001"
    }
  ]
},
{
  "ObjectType": "01",
  "Namespace": "TDOC",
  "ID": "CLS004",
  "Revision": "001",
  "Name": "Car",
  "Status": "Develop",
  "IsAbstract": false,
  "UnitSystem": 3,
  "ClassType": "Application Class",
  "ClassAttributes": [
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP010#001"
    },
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP011#001"
    }
  ]
```

```
    }
  ]
}
```

The first two classes, **CLS002** and **CLS003**, are the property block classes and the third class, **CLS004**, is the application class that references the two property blocks.

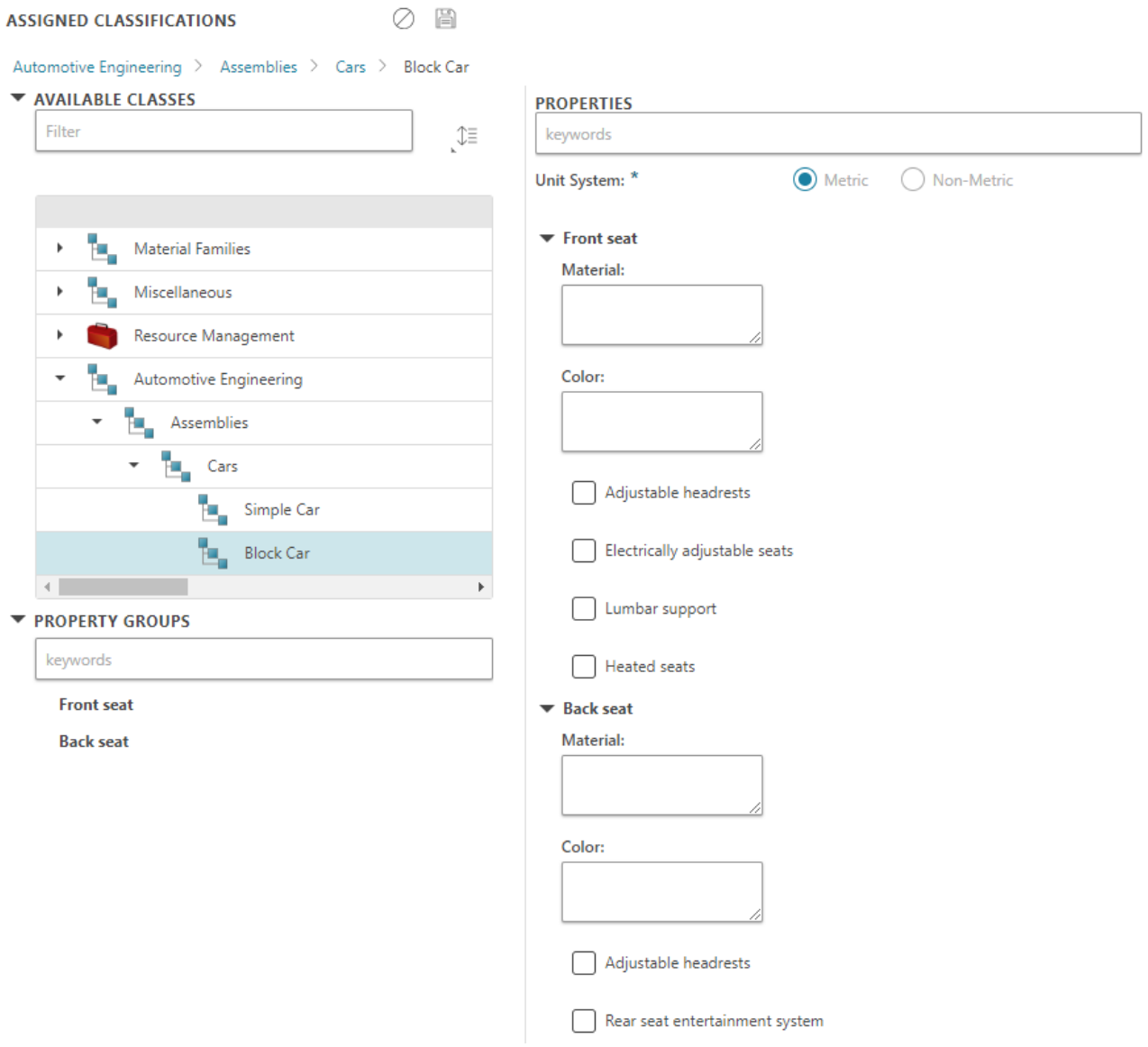
3. Import the node definition that makes the class visible in Active Workspace.

   The **Block car** node must be displayed in the **Cars** node that you created in the first example. To do this, import the new node specifying the **Car** node, **CSTAA1**, as the parent. Save the following in a file named **blockNodeDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "NodeDefinitions": [
    {
      "Namespace": "TDOC",
      "ID": "CSTAA3",
      "Revision": "001",
      "Parent": {
        "Namespace": "TDOC",
        "ID": "CSTAA1",
        "Revision": "001"
      },
      "Name": "Block Car",
      "ApplicationClass": {
        "Namespace": "TDOC",
        "ID": "CLS004",
        "Revision": "001"
      }
    }
  ]
}
```

4. Import the definitions into the database using **Classification Manager** or with **clsutility commands**.

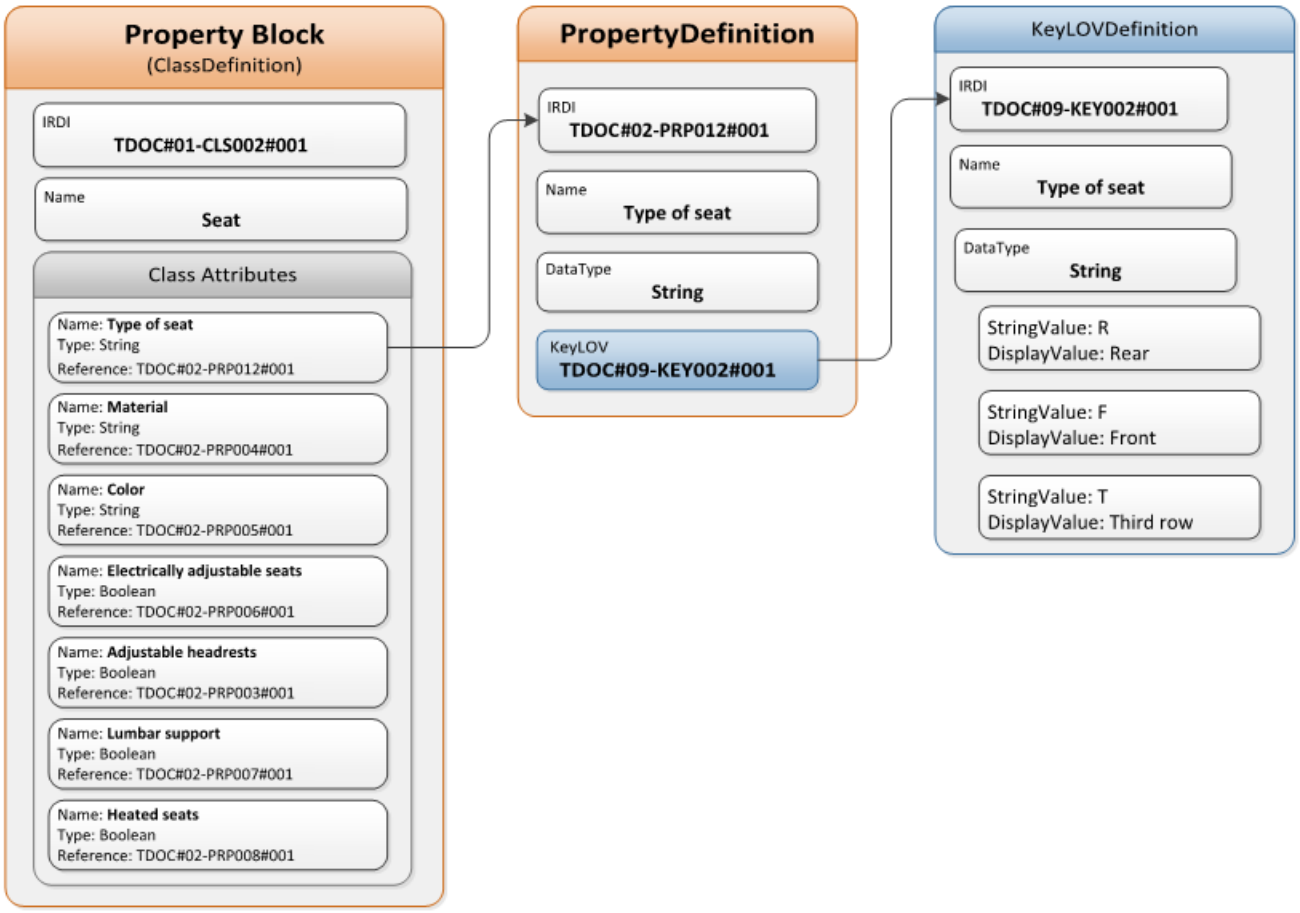
The structure is displayed as follows:



Import a class with cardinal properties

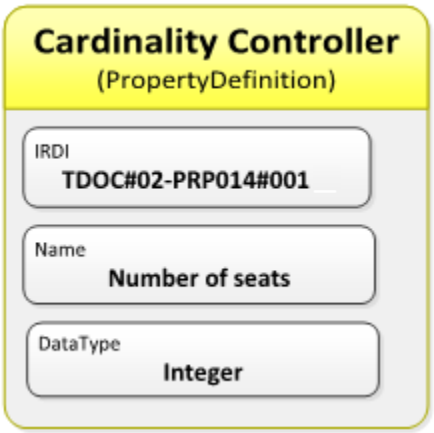
This example shows how to create a class where we can specify the number of seat rows in a car during classification. For example, when classifying a sports car, there is only one row of seats, so only one set of seat properties is necessary. When classifying a family van, there are three rows of seats that each need a set of properties to describe them.

We begin with a set of properties, a property block:

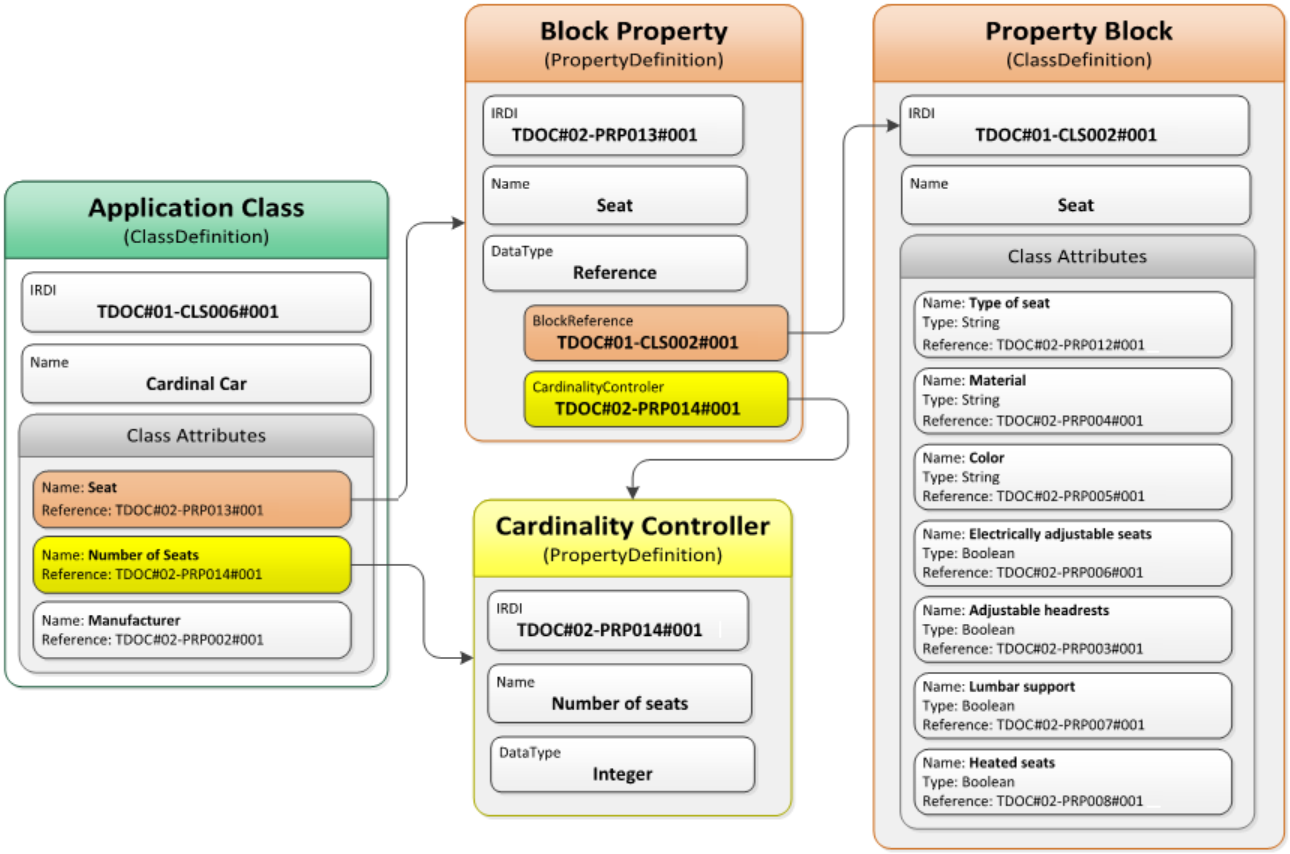


This property block class contains the same attributes as in the previous example, but additionally includes a new key-LOV property, **Type of seat**, where you can specify whether the seats are front, back, or third-row seats.

The number of times that this block of properties is displayed in the user interface is determined by a special property called a *cardinality controller*:



A cardinality controller is referenced by another property and is only of significance if set to **true**. If this is the case, then the user can specify the number of times that the set of properties is displayed.



To create the JSON files required to import these classes and properties:

1. Create the key-LOV definitions.

   This example introduces a new key-LOV, **Type of seat**, with three values, **Front**, **Rear**, and **Third**. Save the following in a file named **cardinalKeyLOVDef.json**

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "KeyLOVDefinitions": [
    {
      "ObjectType": "09",
      "Namespace": "TDOC",
      "ID": "KEY002",
      "Revision": "001",
      "Name": "Type of seat",
      "Status": "Develop",

      "LOVItems": {
```

```
        "DataType": "String",
        "LOVStringItems": [
          {
            "StringValue": "F",
            "DisplayValue": "Front"
          },
          {
            "StringValue": "R",
            "DisplayValue": "Rear"
          },
          {
            "StringValue": "T",
            "DisplayValue": "Third row"
          }
        ]
      }
    }
  ]
}
```

2. Create the individual property definitions.

   There are three new properties in this example: **Type of seat** that references the new key-LOV, **Seat**, the property that specifies the property block class that is to be displayed and acts as an indicator that cardinality is involved, and the **Number of seats** that holds the count.

   Save the following in a file named **cardinalPropDef.json**:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "PropertyDefinitions": [
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP012",
      "Revision": "001",
      "Name": "Type of seat",
      "Status": "Develop",
      "DataType": {
        "Type": "String",
        "KeyLOV": "TDOC#09-KEY002#001"
      }
    },
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP013",
```

```
      "Revision": "001",
      "Name": "Seat",
      "Status": "Develop",
      "DataType": {
        "Type": "Reference",
        "BlockReference": "TDOC#01-CLS005#001",
        "CardinalityController": "TDOC#02-PRP014#001"
      }
    },
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP014",
      "Revision": "001",
      "Name": "Number of seats",
      "Status": "Develop",
      "DataType": {
        "Type": "Integer"
      }
    }
  ]
}
```

3. Create the class definitions, including the property block classes.

   This example introduces two new classes, **CLS005**, the property block class, and **CLS006**, the application class. Save the following in a file named **cardinalClassDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS005",
      "Revision": "001",
      "Name": "Seat properties",
      "Status": "Develop",
      "IsAbstract": true,
      "UnitSystem": 3,
      "ClassType": "Block",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP012#001"
        },
        {
          "Type": "Property",
```

```
        "Reference": "TDOC#02-PRP004#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP005#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP006#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP003#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP007#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP008#001"
      }
    ]
  },
  {
    "ObjectType": "01",
    "Namespace": "TDOC",
    "ID": "CLS006",
    "Revision": "001",
    "Name": "Cardinal car",
    "Status": "Develop",
    "IsAbstract": false,
    "UnitSystem": 3,
    "ClassType": "Application Class",
    "ClassAttributes": [
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP013#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP014#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP002#001"
      }
    ]
  }
```

```
    ]
}
```

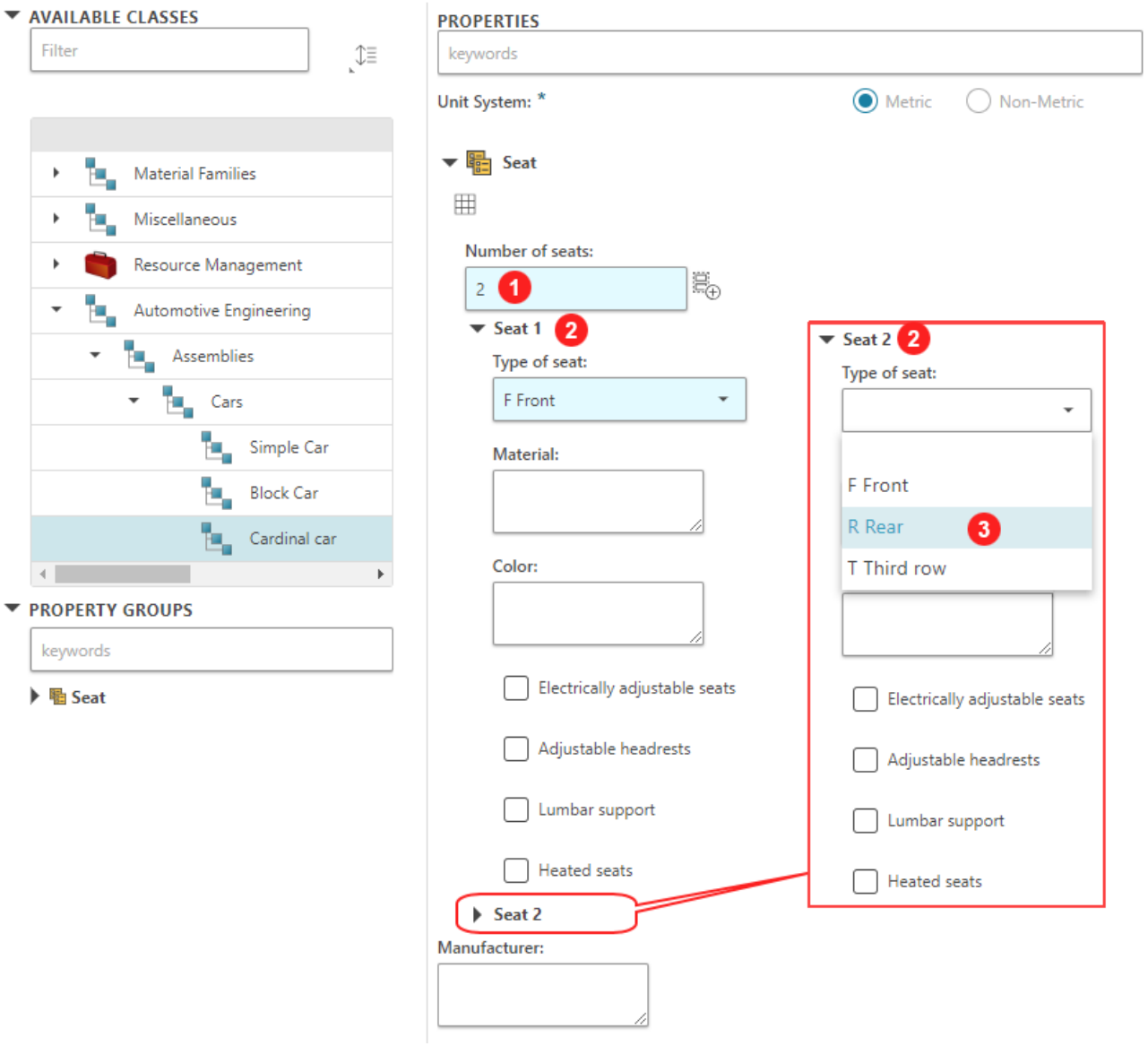
4. Import the node definition that makes the class visible in.

   The **Cardinal car** node must be displayed in the **Cars** node that we created in the first example. To do this, import the new node specifying the **Car** node, **CSTAA1**, as the parent. Save the following in a file named **cardinalNodeDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "NodeDefinitions": [
    {
      "Namespace": "TDOC",
      "ID": "CSTAA4",
      "Revision": "001",
      "Parent": {
        "Namespace": "TDOC",
        "ID": "CSTAA1",
        "Revision": "001"
      },
      "Name": "Cardinal car",
      "ApplicationClass": {
        "Namespace": "TDOC",
        "ID": "CLS006",
        "Revision": "001"
      }
    }
  ]
}
```

5. Import the definitions into the database using **Classification Manager** or with **clsutility commands**.

The structure is displayed as follows:



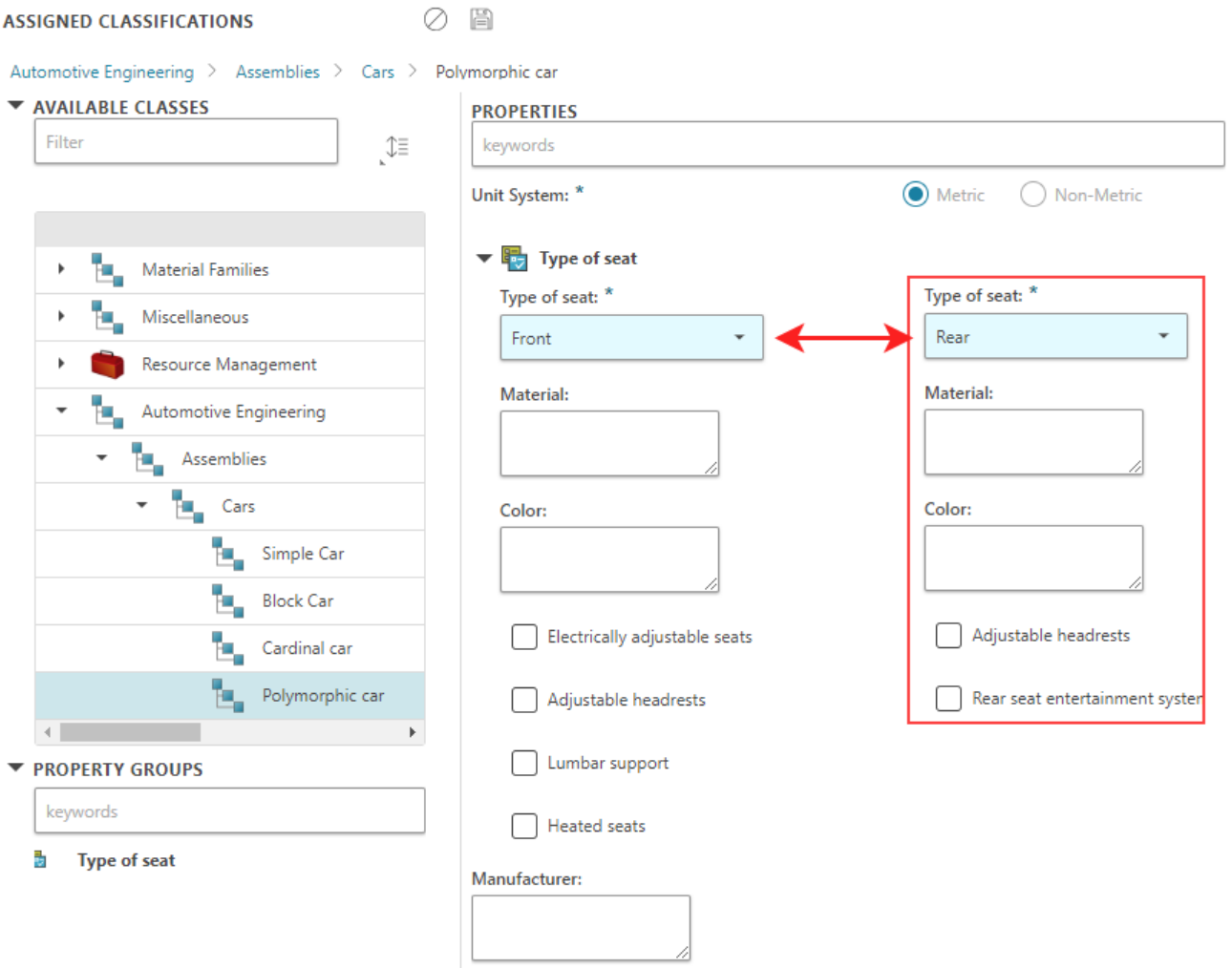
After specifying the number of seats in 1, that number of **Seat** property blocks is displayed 2. You can assign different values for each seat row. With the new key-LOV 3, you can specify a separate type of seat for each row.

Import a class with polymorphic properties

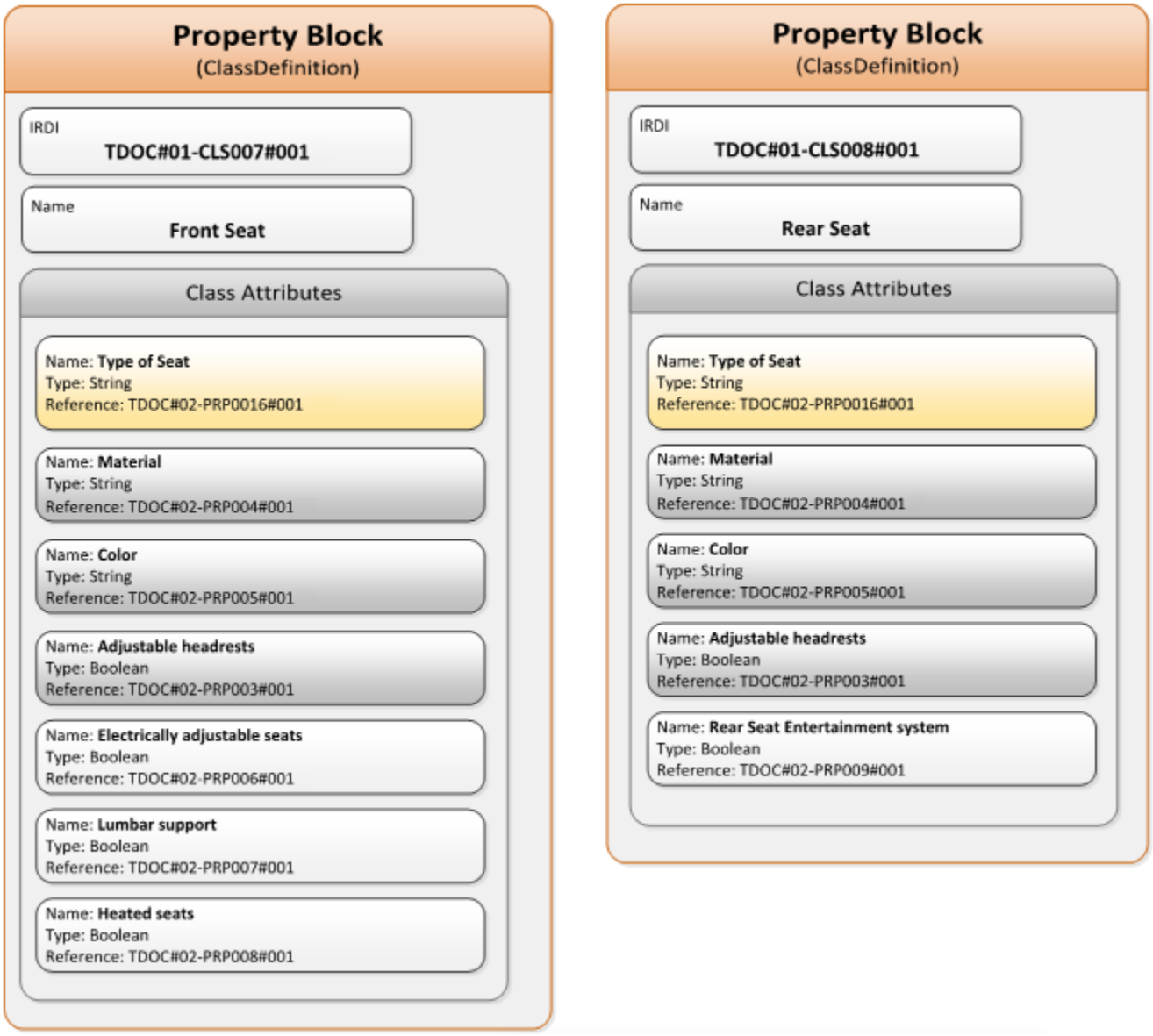
This example shows how to create a class where we can specify the properties displayed depending on the type of seat row in a car that you select during classification. The following table displays the different properties available for each seat row type:

Front seat	Rear seat
Material	Material
Color	Color
Adjustable headrests	Adjustable headrests
Electrically adjustable seats	Rear seat entertainment
Lumbar support	
Heated seats	

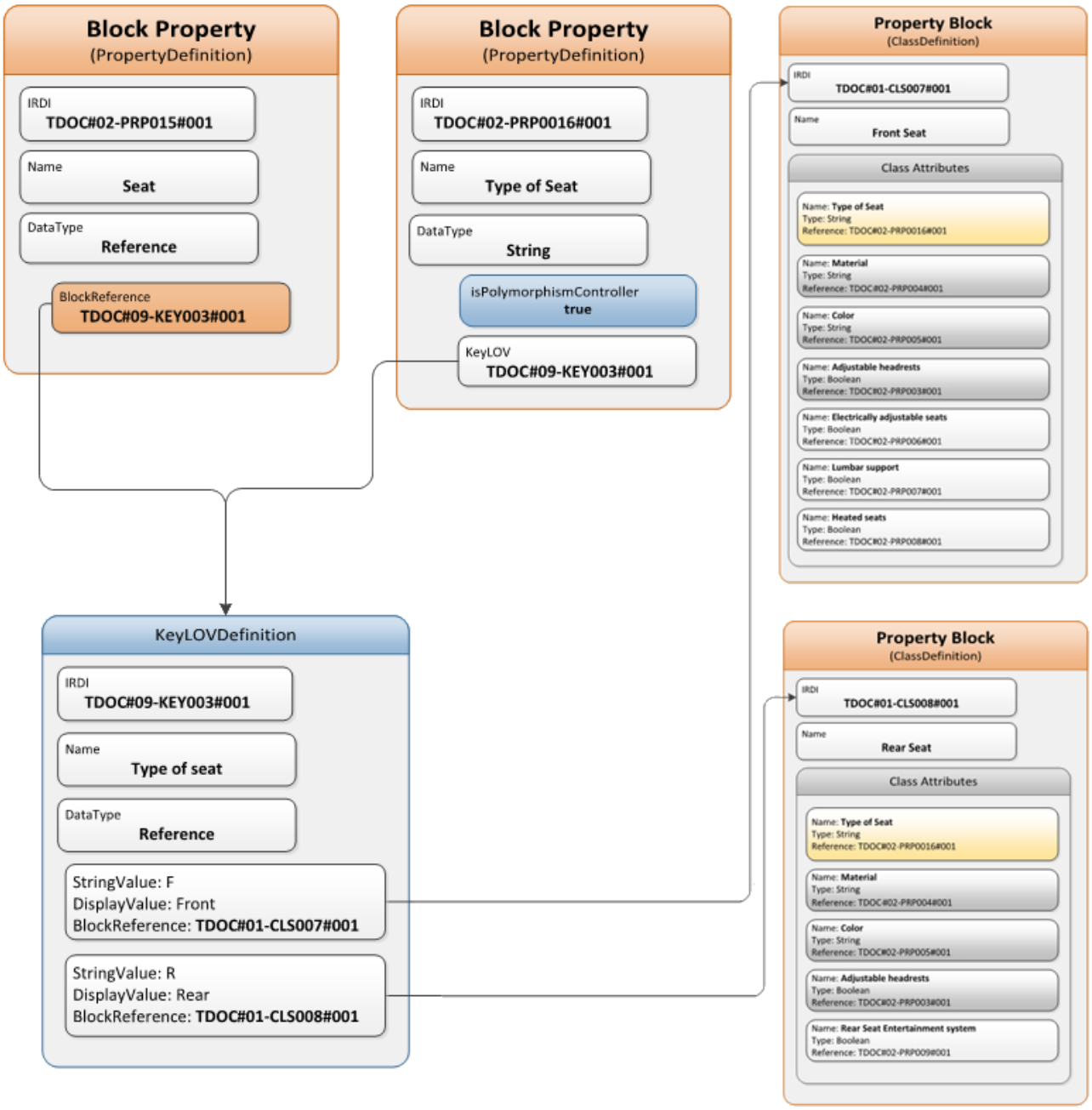
These are the same properties used in the **second example**. The difference is that in this example, you can specify the seat type during classification and, depending on this selection, display different properties. The concept of displaying different properties depending on the value of an additional property is referred to as *polymorphism*.



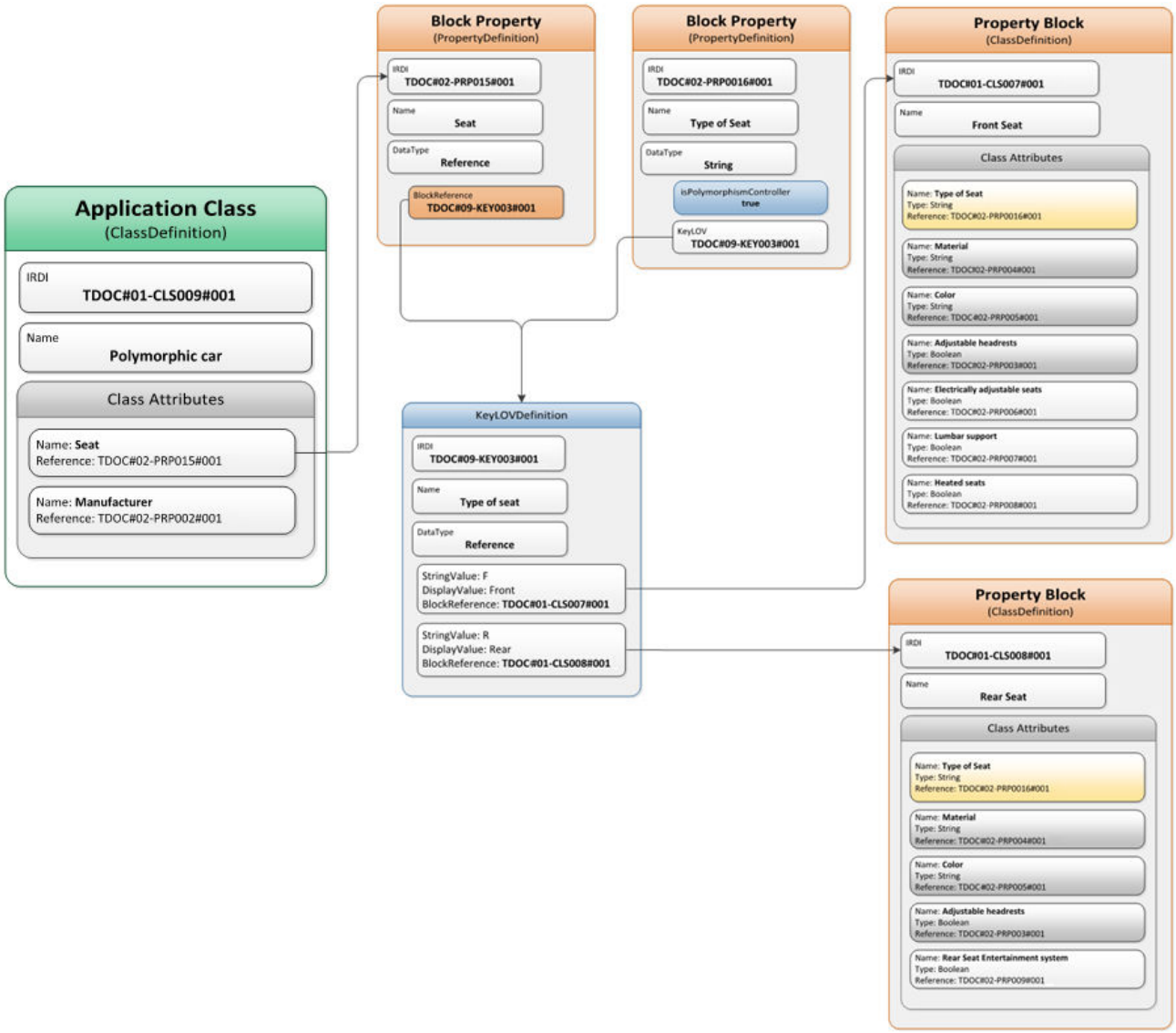
This example contains two classes describing the front and rear seats:



Each of the property block classes contains a special property that specifies the type of seat. This property references a key-LOV with the values **Front** or **Rear**.



The following application class contains the polymorphic **Seat** property that references the key-LOV.



Another property that references the key-LOV, **PRP016**, contains the **isPolymorphismController** attribute. When this property is used in a property block class (**CLS0007** or **CLS008**), the **isPolymorphismController** attribute indicates that the property is polymorphic and that its value determines the list of class properties displayed in the user interface.

To create the JSON files for this example:

1. Create the key-LOV definition.

   This example introduces one new key-LOV that is saved in a file called **polymorphicKeylovDef.json**:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
```

```
  "KeyLOVDefinitions": [
    {
      "ObjectType": "09",
      "Namespace": "TDOC",
      "ID": "KEY003",
      "Revision": "001",
      "Name": "Type of seat",
      "Status": "Develop",
      "LOVItems": {
        "DataType": "Reference",
        "LOVReferenceItems": [
          {
            "StringValue": "F",
            "DisplayValue": "Front",
            "BlockReference": "TDOC#01-CLS007#001"
          },
          {
            "StringValue": "R",
            "DisplayValue": "Rear",
            "BlockReference": "TDOC#01-CLS008#001"
          }
        ]
      }
    }
  ]
}
```

The values of the key-LOV, **Front** and **Rear**, reference their respective property block classes, **CLS007** and **CLS008**.

2. Create the individual property definitions.

   This example uses many of the properties created in the earlier examples, as well as two new properties. Create the following JSON file and save it in a file named **polymorphicPropDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "PropertyDefinitions": [
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP015",
      "Revision": "001",
      "Name": "Seat",
      "Status": "Develop",
      "DataType": {
        "Type": "Reference",
```

```
        "BlockReference": "TDOC#09-KEY003#001"
      }
    },
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP016",
      "Revision": "001",
      "Name": "Type of Seat",
      "Status": "Develop",
      "DataType": {
        "Type": "String",
        "KeyLOV": "TDOC#09-KEY003#001",
        "IsPolymorphismController": true
      }
    }
  ]
}
```

PRP016 contains the **IsPolymorphismController** attribute.

3. Create the class definitions, including the property block classes.

   This example introduces two new property block classes very similar to those in the block class example, but additionally containing the polymorphic **PRP016** property. Save this in a file named **polymorphicClassDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS007",
      "Revision": "001",
      "Name": "Front seat",
      "Status": "Develop",
      "IsAbstract": true,
      "UnitSystem": 3,
      "ClassType": "Block",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP016#001"
        },
        {
          "Type": "Property",
```

```
        "Reference": "TDOC#02-PRP004#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP005#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP006#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP003#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP007#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP008#001"
      }
    ]
  },
  {
    "ObjectType": "01",
    "Namespace": "TDOC",
    "ID": "CLS008",
    "Revision": "001",
    "Name": "Back seat",
    "Status": "Develop",
    "IsAbstract": true,
    "UnitSystem": 3,
    "ClassType": "Block",
    "ClassAttributes": [
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP016#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP004#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP005#001"
      },
      {
        "Type": "Property",
```

```
            "Reference": "TDOC#02-PRP003#001"
          },
          {
            "Type": "Property",
            "Reference": "TDOC#02-PRP009#001"
          }
        ]
      },
      {
        "ObjectType": "01",
        "Namespace": "TDOC",
        "ID": "CLS009",
        "Revision": "001",
        "Name": "Polymorphic car",
        "Status": "Develop",
        "IsAbstract": false,
        "UnitSystem": 3,
        "ClassType": "Application Class",
        "ClassAttributes": [
          {
            "Type": "Property",
            "Reference": "TDOC#02-PRP015#001"
          },
          {
            "Type": "Property",
            "Reference": "TDOC#02-PRP002#001"
          }
        ]
      }
    ]
}
```

The application class **CLS009** references **PRP015**, which is associated with the polymorphic attribute **PRP016** through the key-LOV.

4. Import the node definition that makes the class visible.

   The **Polymorphic car** node must be displayed in the **Cars** node that we created in the first example. To do this, import the new node specifying the **Car** node **CSTAA1** as the parent. Save the following in a file named **polymorphicNodeDef.json**.

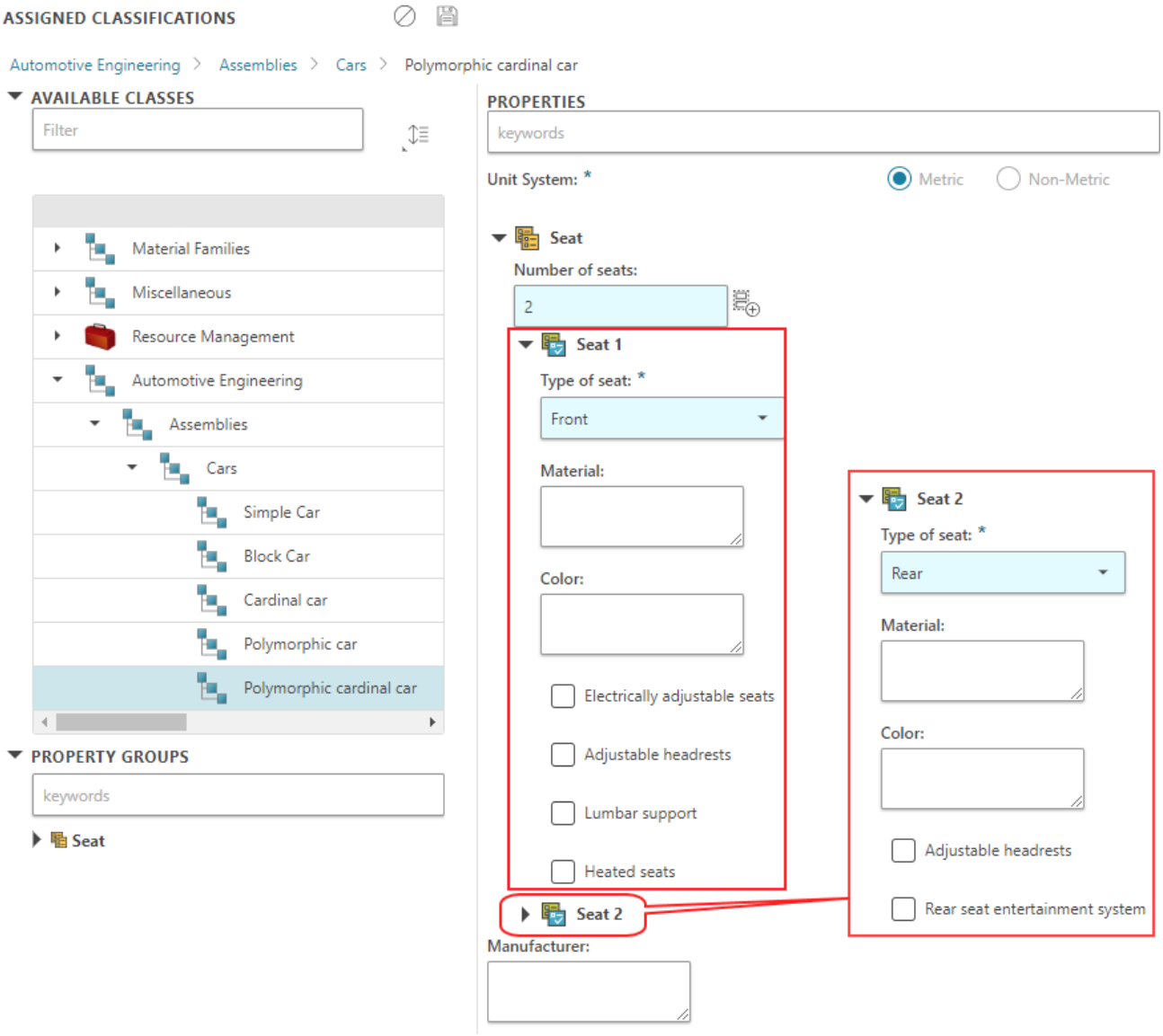
```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "NodeDefinitions": [
    {
      "Namespace": "TDOC",
      "ID": "CSTAA5",
```

```
        "Revision": "001",
        "Parent": {
          "Namespace": "TDOC",
          "ID": "CSTAA1",
          "Revision": "001"
        },
        "Name": "Polymorphic car",
        "ApplicationClass": {
          "Namespace": "TDOC",
          "ID": "CLS009",
          "Revision": "001"
        }
      }
    ]
  }
```

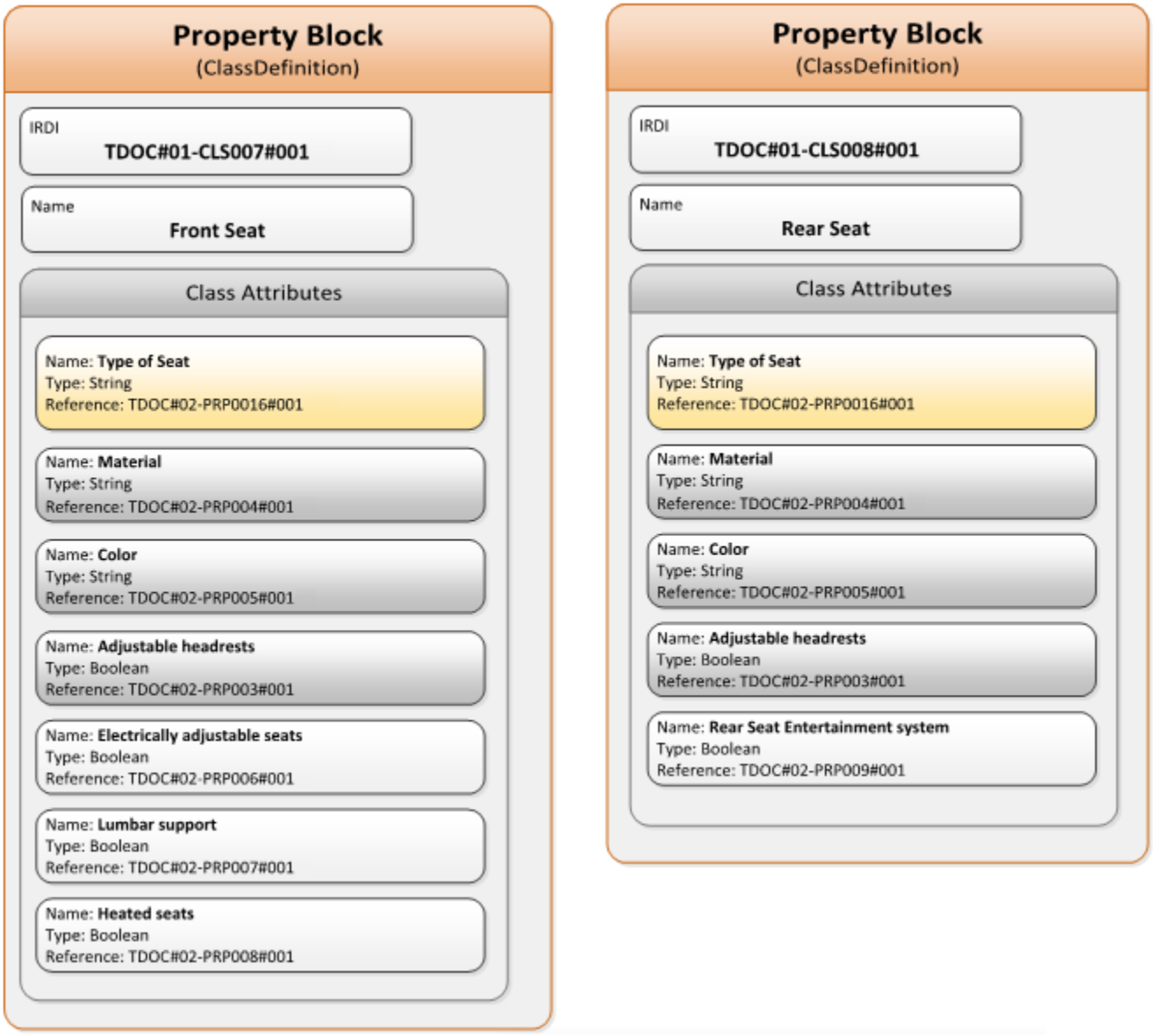
5. Import the definitions into the database using **Classification Manager** or with **clsutility commands**.

Import a class with polymorphic and cardinal properties

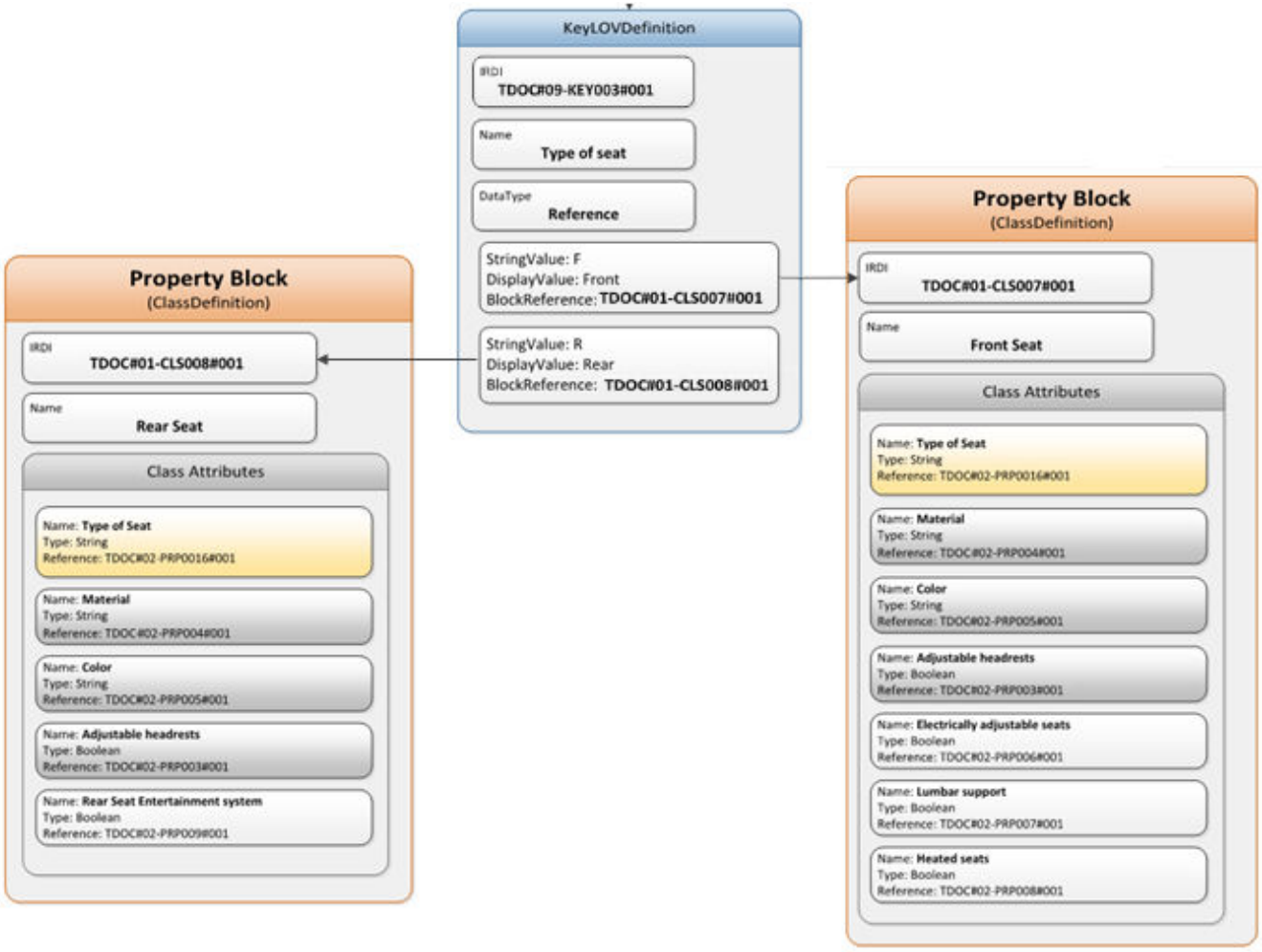
This example shows how to create a class where you can first specify the number of seat rows a car has during classification. For each seat row, the properties displayed depend on the type of seat row that you select.



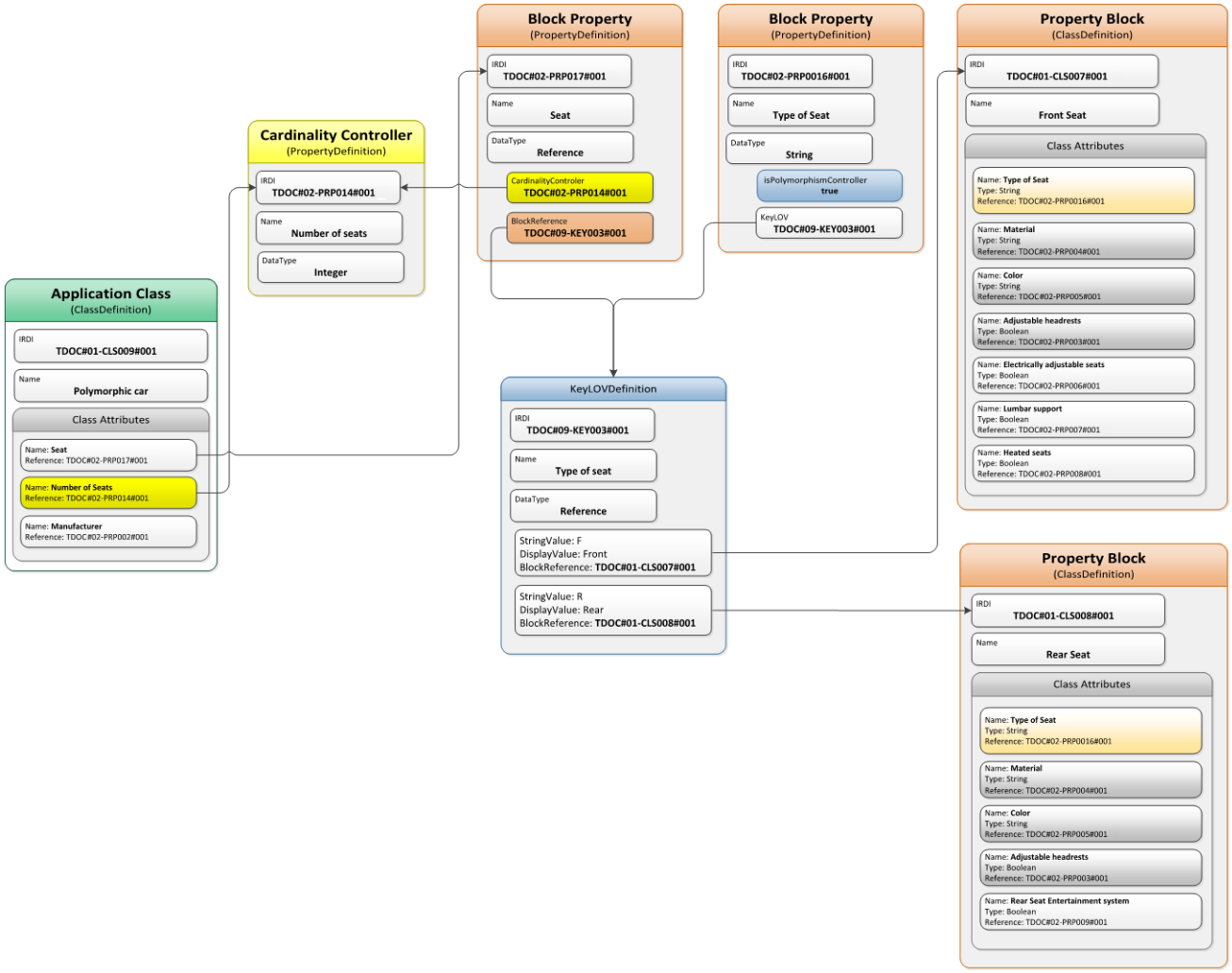
This example uses the same two property block classes as the **previous example**, describing the front and rear seats:



Again, these classes are referenced by the same key-LOV as in the previous example, which determines which of these classes is displayed:



The application class references the properties that determine the polymorphism and cardinality:



After completing the previous examples, there are only a few new objects required to import this example. To create the JSON files for this example:

1. Create the individual property definitions.

   This example uses **PRP014** created in the cardinality example and **PRP016** created in the polymorphism example. One new property is required that references the key-LOV created in the polymorphism example. Create the following JSON file and save it in a file named **poly_cardPropDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "PropertyDefinitions": [
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP017",
```

```
        "Revision": "001",
        "Name": "Seat",
        "Status": "Develop",
        "DataType": {
          "Type": "Reference",
          "BlockReference": "TDOC#09-KEY003#001",
          "CardinalityController": "TDOC#02-PRP014#001"
        }
      }
    ]
  }
```

This property references the key-LOV definition, as well as the cardinality controller property. The polymorphism is achieved by the reference of **PRP016** to the key-LOV.

2. Create the class definitions.

   This example introduces only one new application class. Save this in a file named **poly_cardClassDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS010",
      "Revision": "001",
      "Name": "Polymorphic cardinal car",
      "Status": "Develop",
      "IsAbstract": false,
      "UnitSystem": 3,
      "ClassType": "Application Class",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP014#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP017#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP002#001"
        }
      ]
```

```
      }
    ]
  }
```

3. Import the node definition that makes the class visible.

   The **Polymorphic cardinal car** node must be displayed in the **Cars** node that we created in the first example. To do this, import the new node specifying the **Car** node **CSTAA1** the as parent. Save the following in a file named **poly_cardNodeDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "NodeDefinitions": [
    {
      "Namespace": "TDOC",
      "ID": "CSTAA6",
      "Revision": "001",
      "Parent": {
        "Namespace": "TDOC",
        "ID": "CSTAA1",
        "Revision": "001"
      },
      "Name": "Polymorphic cardinal car",
      "ApplicationClass": {
        "Namespace": "TDOC",
        "ID": "CLS010",
        "Revision": "001"
      }
    }
  ]
}
```

4. Import the definitions into the database using **Classification Manager** or with **clsutility commands**.

Importing classification objects to CST

Classification objects can be imported using the JSON format. The schema file for this is:

..\TD\classification\json\schema\advanced\Classification_Save_PropertyRecords_Request_advanced.schema.json

When creating the JSON file, you can specify whether only the item for the property record is created, or additionally specify a class into which the item is immediately classified during import. Importing the following example classifies an item into the **Polymorphic cardinal car** class that was created in the **import example**.

Tip:

Find the example as text that you can copy **here**.

```
{
    "SchemaVersion": "1.0.0",
    "Locale": "en_US",
    "PropertyRecords": [
        {
            "ID": "029159/A",
            "ObjectType": "PR",
            "ClassDefinition": "TDOC#01-CLS010#001",
            "UnitSystem": 1,
            "ClassifiedObject": {
                "ObjectType": "Item",
                "ClassifyRevision": true,
                "ID": [
                    {
                        "PropertyName": "item_id",
                        "PropertyValue": "029159"
                    }
                ],
                "Properties": [
                    {
                        "PropertyName": "object_name",
                        "PropertyValue": "my_car"
                    }
                ]
            },
            "Properties": [
                {
                    "ID": "TDOC#02-PRP014#001",
                    "Name": "Number of seats",
                    "Value": 2
                },
                {
                    "ID": "TDOC#02-PRP017#001",
                    "Name": "Seat",
                    "Value": [
                        {
                            "Index": 1,
                            "ClassDefinition": "TDOC#01-CLS008#001",
                            "Properties": [
                                {
                                    "ID": "TDOC#02-PRP004#001",
                                    "Name": "Material",
                                    "Value": "Polyester"
                                },
                                {
                                    "ID": "TDOC#02-PRP005#001",
                                    "Name": "Color",
                                    "Value": "Black"
                                },
                                {
                                    "ID": "TDOC#02-PRP003#001",
                                    "Name": "Adjustable headrests",
                                    "Value": true
                                },
                                {
                                    "ID": "TDOC#02-PRP009#001",
                                    "Name": "Rear seat entertainment system",
                                    "Value": true
                                }
                            ]
                        }
                    ]
                },
                {
                    "ID": "TDOC#02-PRP002#001",
                    "Name": "Manufacturer",
                    "Value": "Fancy Car Company"
                }
            ]
        }
    ]
}
```

In the JSON file:

- Lines 10-25 create the classified item with ID **029159** and name **my_car**.
- Line 30 creates 2 instances (a cardinality of 2) of **Number of seats** and these seats are described in lines 36-99.
- The index shown in line 37 and 73 introduces each of the new seat instances.
- Lines 39 and 75 begin the list of properties for each type of seat.
- The list of properties specific to each seat type is described in lines 40-71 and lines

The **Name** property is not mandatory and is added to this example for purposes of clarity.

To import this property record that is saved in a file called **poly_car_prop_rec.json**, you must use the **clsutility**:

```
clsutility -create -classification_objects
-request="poly_car_prop_rec.json"
```

JSON classification object input example

```
{
    "SchemaVersion": "1.0.0",
    "Locale": "en_US",
    "PropertyRecords": [
        {
            "ID": "029159/A",
            "ObjectType": "PR",
            "ClassDefinition": "TDOC#01-CLS010#001",
            "UnitSystem": 1,
            "ClassifiedObject": {
                "ObjectType": "Item",
                "ClassifyRevision": true,
                "ID": [
                    {
                        "PropertyName": "item_id",
                        "PropertyValue": "029159"
                    }
                ],
                "Properties": [
                    {
                        "PropertyName": "object_name",
                        "PropertyValue": "my_car"
                    }
                ]
            },
            "Properties": [
                {
                    "ID": "TDOC#02-PRP014#001",
                    "Name": "Number of seats",
                    "Value": 2
```

```
},
{
    "ID": "TDOC#02-PRP017#001",
    "Name": "Seat",
    "Value": [
        {
            "Index": 1,
            "ClassDefinition": "TDOC#01-CLS007#001",
            "Properties": [
                {
                    "ID": "TDOC#02-PRP004#001",
                    "Name": "Material",
                    "Value": "Leather"
                },
                {
                    "ID": "TDOC#02-PRP005#001",
                    "Name": "Color",
                    "Value": "Red"
                },
                {
                    "ID": "TDOC#02-PRP006#001",
                    "Name": "Electrically adjustable seats",
                    "Value": true
                },
                {
                    "ID": "TDOC#02-PRP003#001",
                    "Name": "Adjustable headrests",
                    "Value": true
                },
                {
                    "ID": "TDOC#02-PRP007#001",
                    "Name": "Lumbar support",
                    "Value": true
                },
                {
                    "ID": "TDOC#02-PRP008#001",
                    "Name": "Heated seats",
                    "Value": true
                }
            ]
        },
        {
            "Index": 2,
            "ClassDefinition": "TDOC#01-CLS008#001",
            "Properties": [
                {
                    "ID": "TDOC#02-PRP004#001",
                    "Name": "Material",
                    "Value": "Polyester"
                },
                {
                    "ID": "TDOC#02-PRP005#001",
                    "Name": "Color",
                    "Value": "Black"
                },
                {
                    "ID": "TDOC#02-PRP003#001",
                    "Name": "Adjustable headrests",
                    "Value": true
                },
```

```
                                {
                                    "ID": "TDOC#02-PRP009#001",
                                    "Name": "Rear seat entertainment system",
                                    "Value": true
                                }
                            ]
                        }
                    ]
                },
                {
                    "ID": "TDOC#02-PRP002#001",
                    "Name": "Manufacturer",
                    "Value": "Fancy Car Company"
                }
            ]
        }
    ]
}
```

clsutility commands required to import example json files

When importing files for **this example**, if you choose to import using the **clsutility**, run each of the following commands in a Teamcenter command window:

Import a class with simple properties

```
clsutility -u=user -p=password -g=group -create -keylov_definitions
-request="simpleKeylovDef.json"
clsutility -u=user -p=password -g=group -create -property_definitions
-request="simplePropDef.json"
clsutility -u=user -p=password -g=group -create -class_definitions
-request="simpleApplicationClassDef.json"
clsutility -u=user -p=password -g=group -create -node_definitions
-request="simpleNodeDef.json"
```

Import a class with a block properties

```
clsutility -u=user -p=password -g=group -create -property_definitions
-request="blockPropDef.json"
clsutility -u=user -p=password -g=group -create -class_definitions
-request="blockClassDef.json"
clsutility -u=user -p=password -g=group -create -node_definitions
-request="blockNodeDef.json"
```

Import a class with cardinal properties

```
clsutility -u=user -p=password -g=group -create -keylov_definitions
-request="cardinalKeyLOVDef.json"
clsutility -u=user -p=password -g=group -create -property_definitions
-request="cardinalPropDef.json"
clsutility -u=user -p=password -g=group -create -class_definitions
-request="cardinalClassDef.json"
clsutility -u=user -p=password -g=group -create -node_definitions
-request="cardinalNodeDef.json"
```

Import a class with polymorphic properties

```
clsutility -u=user -p=password -g=group -create -keylov_definitions
-request="polymorphicKeylovDef.json"
clsutility -u=user -p=password -g=group -create -property_definitions
-request="polymorphicPropDef.json"
clsutility -u=user -p=password -g=group -create -class_definitions
-request="polymorphicClassDef.json"
clsutility -u=user -p=password -g=group -create -node_definitions
-request="polymorphicNodeDef.json"
```

Import a class with polymorphic and cardinal properties

```
clsutility -u=user -p=password -g=group -create -property_definitions
-request="poly_cardPropDef.json"
clsutility -u=user -p=password -g=group -create -class_definitions
-request="poly_cardClassDef.json"
clsutility -u=user -p=password -g=group -create -node_definitions
-request="poly_cardNodeDef.json"
```

Import a property record

```
clsutility -create -classification_objects
-request="poly_car_prop_rec.json"
```

Export classified data for advanced classification

You can export classified data in the BMEcat XML from the user interface.

1. Open the classified data that you want to export.

2. Choose **More Commands** ••• > **Import/Export** > **Export Classification Data**.

3. In the **File Name** box, give a name to the file that will be downloaded.

4. Click **Export**.

When the export is complete, you are notified in the **Alert** area where you can view the details of the export.

The file is available in the **Target Object** section of the alert report. Download the file from that section.

This feature requires that Subscription Manager is running. If the Subscription Manager is not running, no notification is displayed in the **Alert** area, but the operation still occurs in the background.

Export classified objects using the clsutility command

To export the classification information (the *ICO*) for an object classified in a CST hierarchy, use the following **clsutility** command:

clsutility -u=*user* **-p=***password***-g=***group* **-find -classification_objects -request=***JSON_file* **-output=***path-to-output-file*

This command requires a JSON file resembling the following (see the **Classification_FindRequest_advanced.schema.json** schema file):

```
{
    "SchemaVersion": "1.0.0",
    "Locale" : "en_US",
    "ObjectIDList": [
        {
            "ID": "026144/A"
        }
    ]
}
```

The input JSON file contains a list of object IDs for which the ICO information is exported to a file of your choice.

Organizing classes with hierarchical positions in ECLASS

The ECLASS standard uses the concept of a *class code* or *coded name* to uniquely identify classes and designate the order in which classes and subclasses are displayed. The coded name consists of a group of four two-digit numbers, separated by a hyphen. Each of these numbers represents a position in the hierarchy.

The ECLASS standard assumes a four-level hierarchy, with the fourth level always consisting of leaf classes. Properties are assigned to the fourth level only. If this is the structure of your classification hierarchy, then you can use the coded name, or the *hierarchical position* to sort the classes in the classification hierarchy. The hierarchical position is then honored by Teamcenter at import of the JSON file. According to the ECLASS standard (see the section on *Blocked Class Codes* in the ECLASS documentation), you can use the number 98 and 99 to embed custom classes into the ECLASS hierarchy.

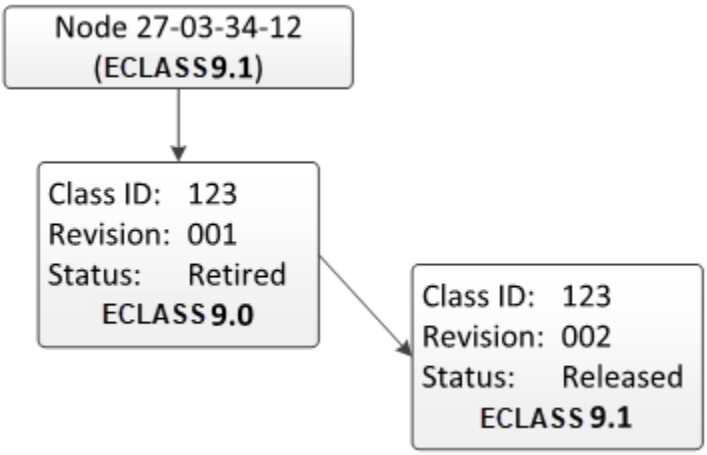
> Caution:
>
> If your hierarchy does not follow this four-level pattern, do not use the **Hierarchical position** option in your data definitions.

Managing multiple ECLASS releases

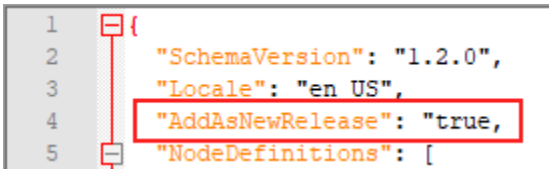
The ECLASS hierarchy is available in multiple releases. You can import and access in Teamcenter where you can classify objects in different releases of the classification hierarchy simultaneously.

To facilitate having multiple releases of a hierarchy, the concept of a *revision chain* is used. The following scenario takes place during the import of ECLASS releases.

You import the ECLASS 9.0. Later, you decide to update to the ECLASS 9.1 release Because you cannot have two classes with a **Released** status at the same time, the original class referenced by **Node 27-03-34-12** is set to **Retired** when you **set the new class to Released**. Even though the new 9.1 **Node 27-03-34-12** still points to **Revision 001** of the class, internally, the node now references **Revision 002** of the class.

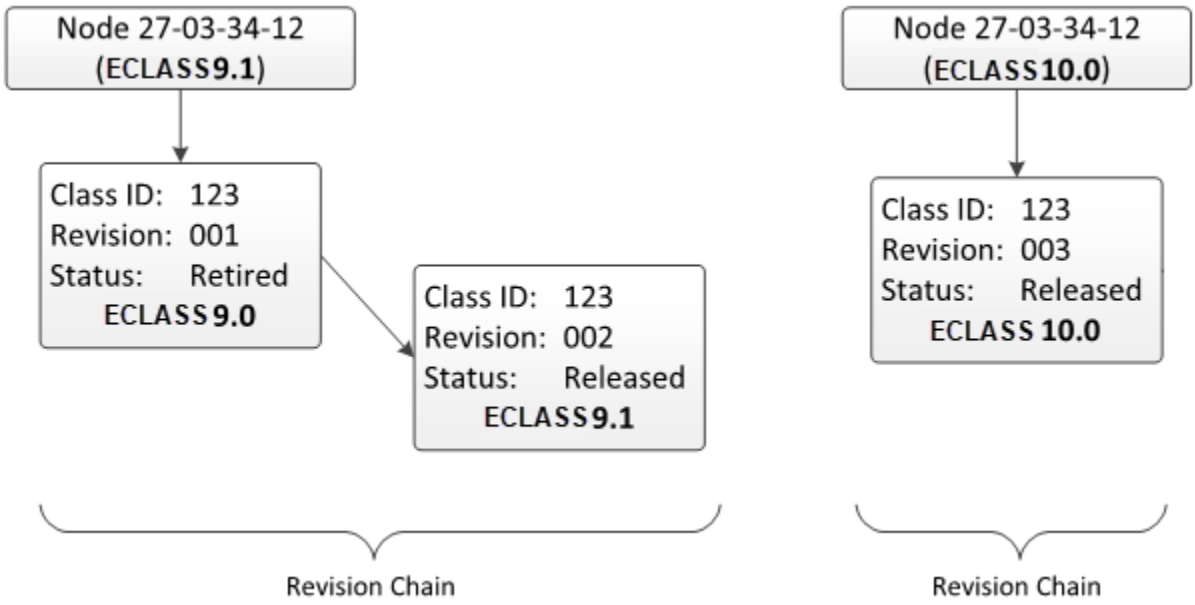


Now your company wants to use the new ECLASS 10.0 release. You still, however, want to classify in (or at least see) the 9.1 hierarchy at the same time. To do this, add the **AddasNewRelease** option to the JSON files that you import (requires at least schema version 1.2.0):



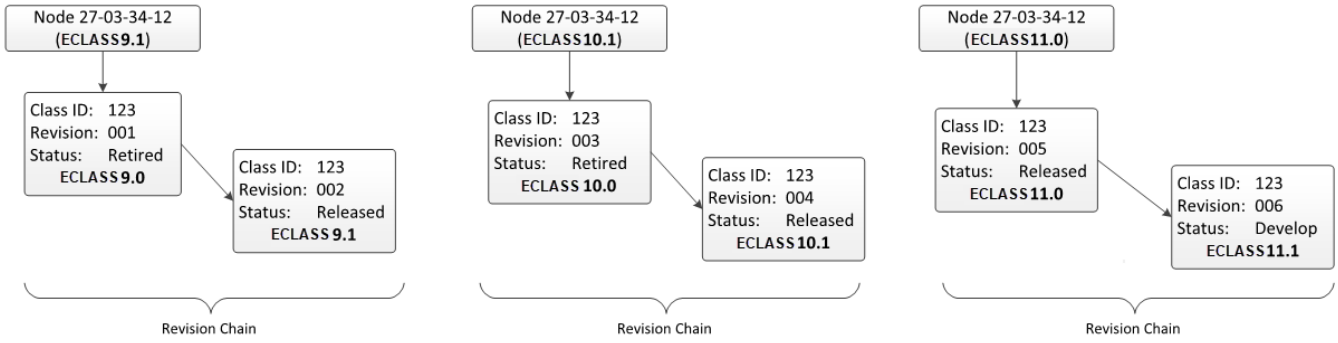
> Tip:
>
> You can add this to a JSON definition when you **convert the hierarchy definitions from ontoML with the eclass2json utility**.

Importing a JSON definition with the **AddasNewRelease** option set to **true** creates a new revision chain.

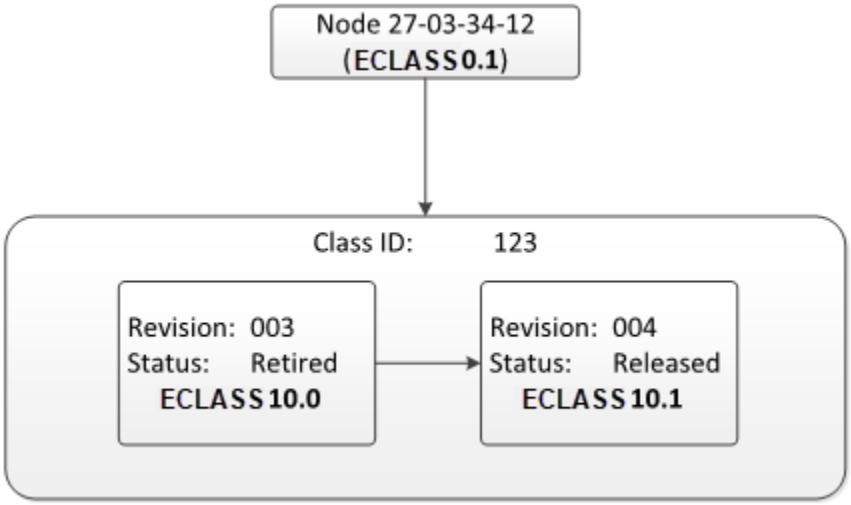


There are now two revisions of a class in the **Released** state but they are in two different revision chains so that users can classify in both 9.1 and 10.0 classes simultaneously.

When you continue upgrading and importing new ECLASS releases, the revision chains may look like this:



Note Revision 006 of the class that was imported with the ECLASS 11.1 release is still in **Develop** state (you haven't yet set it to **Released** with the **clsutility**). Therefore, the node still references **Revision 005** of the class. In other words, a node always references the most recent (in terms of revision number) class in a **Released** state. That class is used for classification.



The fact that a different revision than the one referenced by the node is found when classifying objects is referred to as *imprecise* handling.

To differentiate releases of the hierarchy when working in multiple releases, the schema file (version 1.2.0) contains an optional **SourceStandard** entry. This option in the JSON file is used to specify the hierarchy version for an administrative object. The **SourceStandard** value is mapped to a display name for the release in the **CST_supported_eclass_releases** site preference. The preference values are set in pairs. Examples of possible values are:

0173-1#11-ECLASS9#001

ECLASS 9.0

0173-1#11-ECLASS91#001

ECLASS 9.1

0173-1#11-ECLASS10.0.1#001

ECLASS 10.0.1

0173-1#11-ECLASS10.1#001

ECLASS 10.1

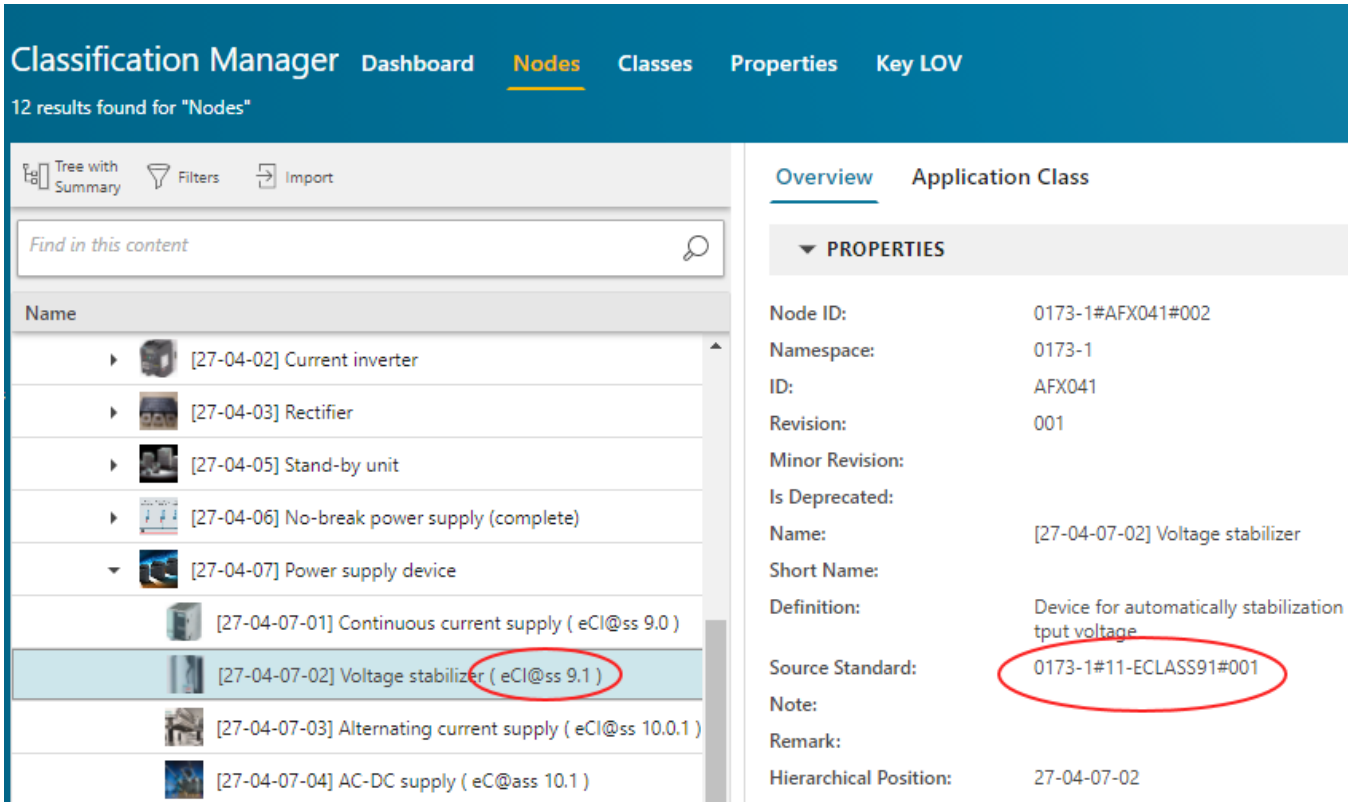
0173-1#11-ECLASS11#001

ECLASS 11.0

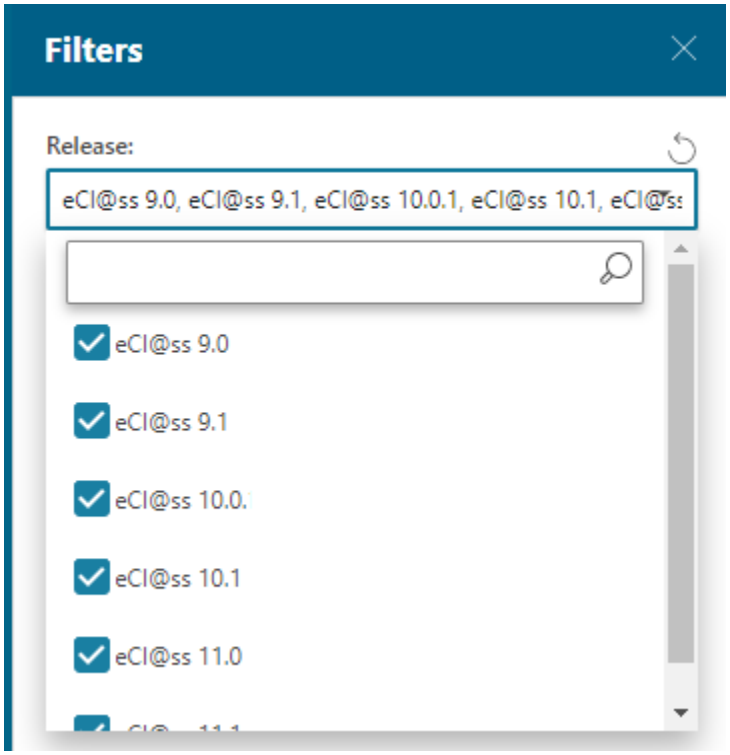
0173-1#11-ECLASS11.1#001

ECLASS 11.1

The first entry represents the IRDI that must be added to the **SourceStandard** entry in the JSON file describing the admin objects. The second entry is the display name shown in parentheses in the classification hierarchy.



If the **CST_supported_eclass_releases** preference contains any IRDI and display name pairs, a **Release** filter is available showing the display names in the preference. You specify which release is visible in this filter:



By default, all releases are selected. If you select one release only, no release is displayed in the classification hierarchy.

When classifying in a system with multiple hierarchies, filters can be set to specify which releases of the hierarchy to display.

Although this mechanism is intended for use with multiple ECLASS hierarchies, it can be used for any type of hierarchy revisions your company may work with. Specify your own **SourceStandard**/display name pairs for the preference value and add your source standard value to the JSON hierarchy objects.

Converting between XML and JSON file format

Converting files between ECLASS XML formats and JSON

ECLASS data is generally delivered in two formats:

- ontoML XML

  Hierarchy (segment) data downloaded from the ECLASS website is delivered in this format.

- BMEcat XML

  Classification objects that are downloaded from the Siemens Industry Mall website are delivered in this format.

For more information, see **https://eclass.eu/support**.

There are several scenarios when importing and exporting classification data that involve converting files from XML formats to JSON and from JSON to BMEcat XML.

Scenario 1: You want to import hierarchy definitions into the database:

1. Download ECLASS hierarchy data (ontoML XML) from the ECLASS website.

   This data contains hierarchy and class definitions only.

2. Convert this data into JSON using the **eclass2json** utility.

3. Import the JSON files into the database.

4. Create objects and classify them using the new classification classes.

Scenario 2: You want to import classified product data in BMEcat XML format into the database:

1. Convert the BMEcat files to JSON format using the **eclass2json** utility.

2. Import the JSON files into your database.

Scenario 3: You have classified data in your classification hierarchy that you want to share with an external partner who also uses ECLASS:

1. Export the classification object (ICO) using **clsutility**.

2. Convert the ICO from JSON to BMEcat XML using the **eclass2json** utility. The BMEcat files include references to any attachments.

Convert hierarchy definitions to JSON format

To import ECLASS hierarchy definitions into Teamcenter, they must be converted into the JSON file format. Use the **eclass2json** utility to perform these conversions.

The utility functions by outputting temporary files that it then references to create the JSON or BMEcat files.

For this reason, the correct creation of the JSON files requires running the utility to output objects individually in the following sequence:

Sequence	Object	Utility argument	Comment
1	Units	**unitsmap**	Convert these once per ECLASS release.
2	Hash maps (temporary files)	**hashmaps**	Convert these once per ECLASS segment.
3	Definitions	**definitions**	Convert these as required. If necessary, you can convert individual object types using the following key words (and in the following order): **keylovs** **properties** **aspects** **blocks** **applicationClasses** **classificationClasses**

The utility requires a specific directory structure that it uses to find its input, as well as to store the completed JSON files. As input, this directory must contain the extracted XML files to be converted. During processing, the utility creates subdirectories for both the temporary hash map files and the newly created JSON files.

1. Obtain ontoML XML hierarchy data from the ECLASS website and extract the contents into a directory.

2. Open a command window and navigate to the directory containing the units file, for example, **eClass9_1_UnitsML_EN_DE**.

3. Call the **eclass2json** Perl script to create the unit hashmap file. The utility requires the unit XML file for input, for example:

```
%TC_PERL% %TC_BIN%\eclass2json\eclass2json.pl -inputFile:eClass9_1_UnitsML_EN_DE.xml
-objectType:unitsmap
```

   The utility creates the units hashmap text file in a subdirectory of the **e2j_output** directory.

4. Navigate to the directory containing the dictionary objects, for example, **eClass9_1_Dictionary_ADVANCED_XML_EN**.

5. Call the **eclass2json** Perl script to create the hashmap file. The utility requires the dictionary XML file for input, for example:

```
%TC_PERL% %TC_BIN%\eclass2json\eclass2json.pl -inputFile:eClass9_1_ADVANCED_EN_SG_13.xml
-objectType:hashmaps
```

   The utility creates the hashmap text files in the **e2j_output** directory.

6. In the same directory, using the same input file, convert the definitions:

```
%TC_PERL% %TC_BIN%\eclass2json\eclass2json.pl -inputFile:eClass9_1_ADVANCED_EN_SG_13.xml
-objectType:definitions
```

   The **e2j_output** directory now contains a subdirectory with the converted JSON files.

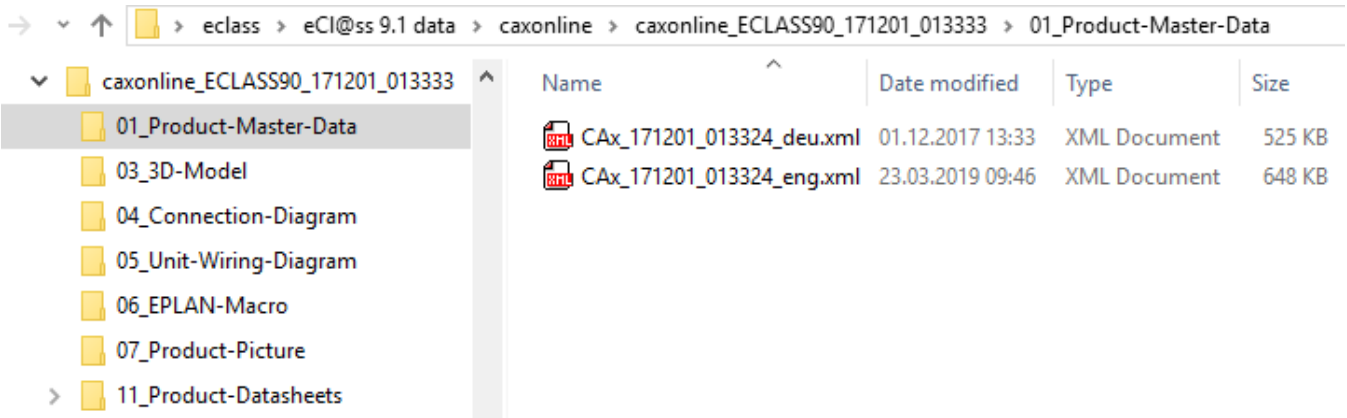
7. Import the newly created JSON files into the database as described in *Importing JSON files using the clsutility command*.

The class hierarchy is now imported and ready for you to classify objects or **import classified objects**.

Convert BMEcat format to JSON

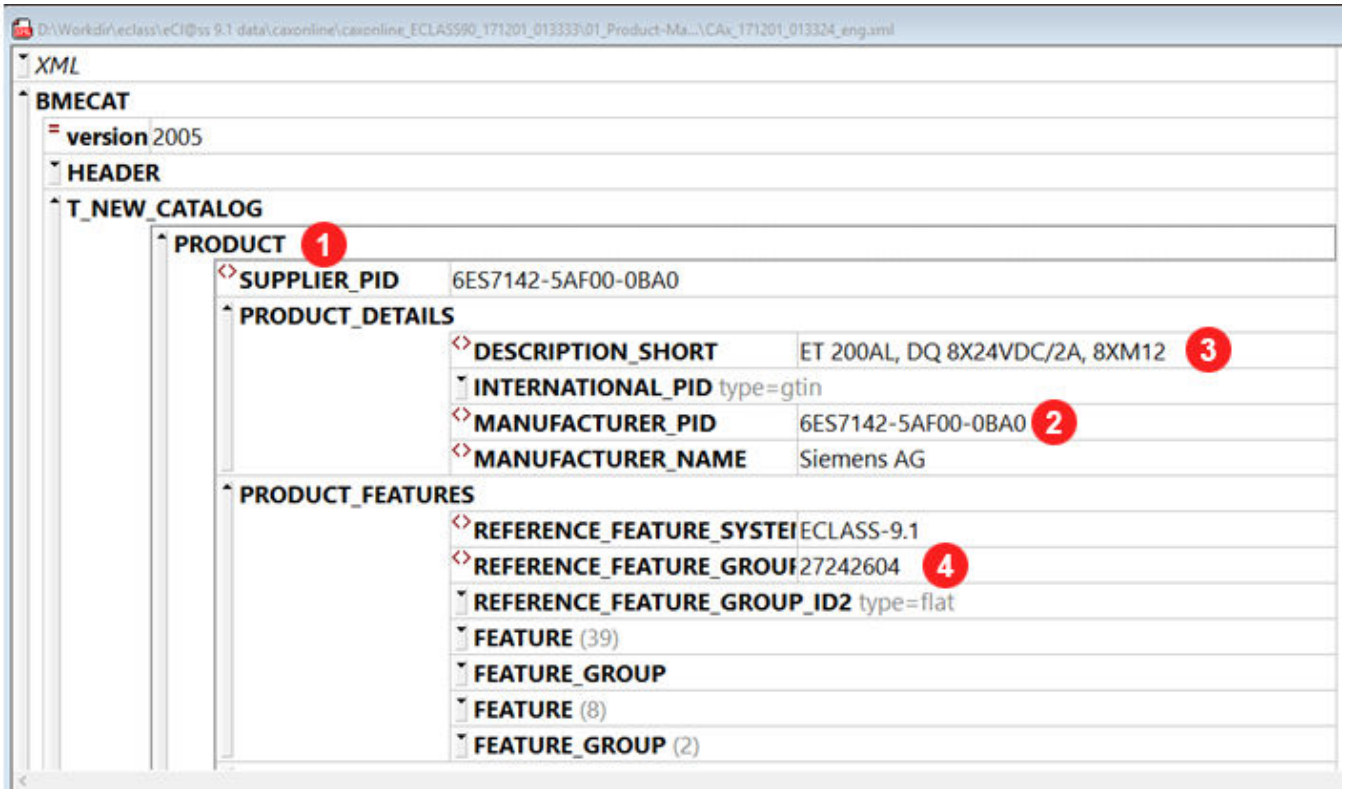
Many companies use the ECLASS standard to classify their product data. This data is delivered s CAx data in the BMEcat XML file format. The files of interest for the **eclass2json** conversion utility are found in the **01_Product-Master-Data** directory of the download package.

The following describes manually converting BMEcat XML files after which you can import them using the **clsutility**. Alternatively, you can **import these files automatically in the user interface**.

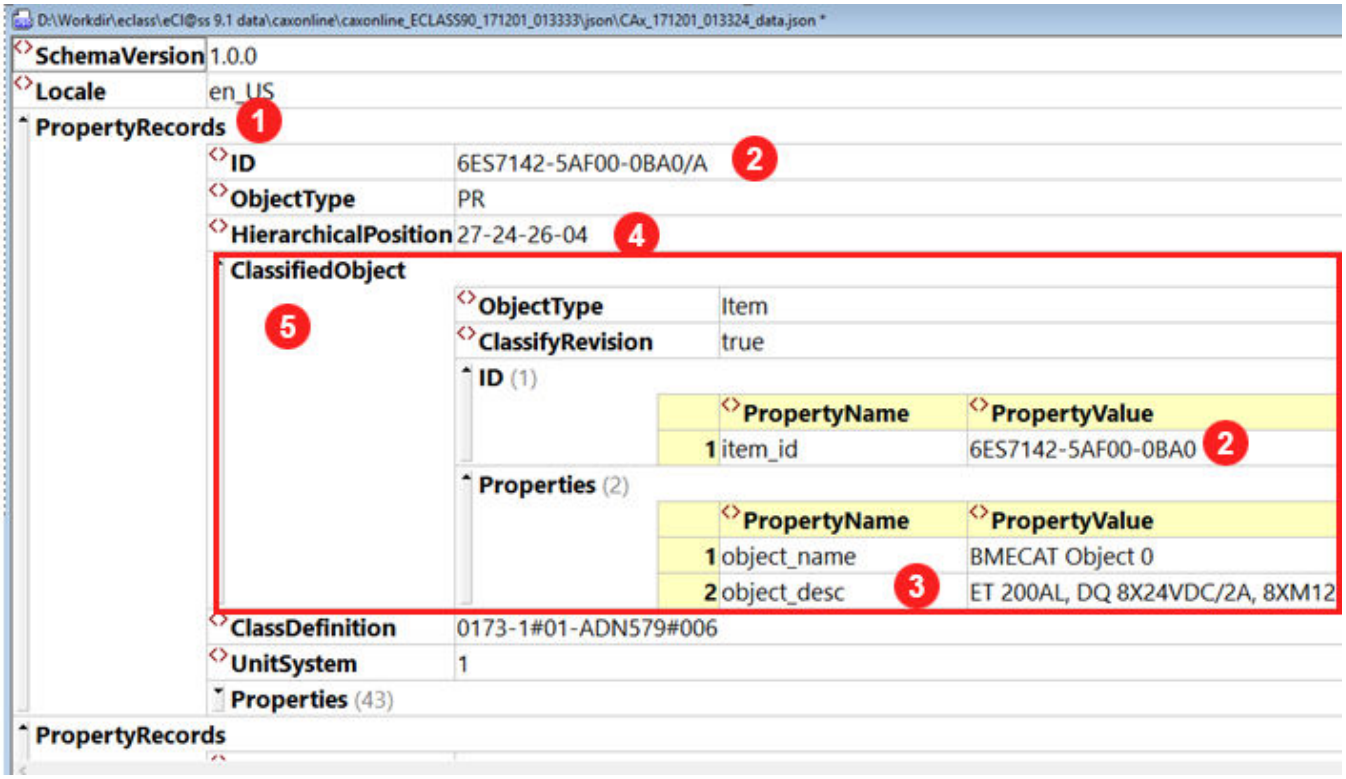


During conversion, the **eclass2json** utility creates a classification object for each **PRODUCT** in the XML file (1). The **FEATURES** are used to create properties. The **MANUFACTURER_PID** is used to create the classification object ID (2).

BMEcat XML file:



JSON file:



The **REFERENCE_FEATURE_GROUP** 4 is used to create the hierarchical position. The first two digits, in this case, **27**, indicate the segment being converted.

The utility creates a section in the JSON file for a classified object 5 and uses the **MANUFACTURER_PID** to identify the item ID. During import, the utility checks whether there is an item revision in the database with that ID. If so, it is automatically classified. If not, the item revision is created and automatically classified. The value of **DESCRIPTION_SHORT** is used as the classification object description 3.

1. Obtain BMEcat XML product data and extract the contents into a directory, *extraction-directory*.

   The extraction generally results in a directory structure with many sub-directories, each containing BMEcat files and reference folders.

2. Copy the following file to the *extraction-directory* location and unzip it.

   TC_DATA/classification/eclass2json/eclass2jsonHashMaps.zip

   A new *hashmaps* directory is created.

3. Open a Teamcenter command window and call the **eclass2json** Perl script:

```
%TC_PERL% %TC_BIN%/eclass2json/eclass2json.pl -DIR "extraction-directory"
-inputFile: "BMEcat-file-path-relative-to-extraction-location" -objectType:data
```

The script creates a **e2j_output** output directory in the directory where the input files are stored. The output directory contains the newly created JSON files.

4. Import the JSON files into the database as described in *Importing JSON files using the clsutility command*.

Convert JSON files to BMEcat to share classified items with suppliers

Teamcenter exports classified product data, also referred to as classification objects (ICOs), in JSON format. To share these ICOs with an external system that uses BMEcat XML format, the JSON files must be converted.

1. Export the desired ICOs using **clsutility** as follows:

   clsutility -find -classification_objects request=*JSON-input-file*

   JSON-input-file lists the ICO object IDs to be exported. The object IDs must be listed with the revision, for example, 123456/A. The JSON file must follow the format specified by the **Classification_FindRequest_advanced.schema.json** schema file found in the **TD\classification\json\schema\advanced** directory.

   The **clsutility** creates a JSON file with the ICO information.

   > Caution:
   >
   > The content of the JSON file created by the **clsutility** may vary slightly in format to that required by the **eclass2json** utility as input. In this case, the JSON file must be manually modified.

2. Open a command window and navigate to the directory containing the BMEcat source files.

3. Run the **eclass2json** utility with the **-objectType:json2BMEcat** argument. For example:

```
%TC_PERL% %TC_BIN%\eclass2json\eclass2json.pl -objectType:json2BMEcat
```

The BMEcat files are stored in the **e2j_output** directory found in the same directory as the input file. If there are attachments, these are also found there.

About the eclass2json properties file

The **eclass2json.properties** file allows you to configure the behavior of the conversions between the following formats:

- ontoML XML to JSON
- BMEcat XML to JSON
- JSON to BMEcat XML

> Caution:
>
> Always create a backup copy of the **eclass2json.properties** file before you modify it in case you must revert to the default values.

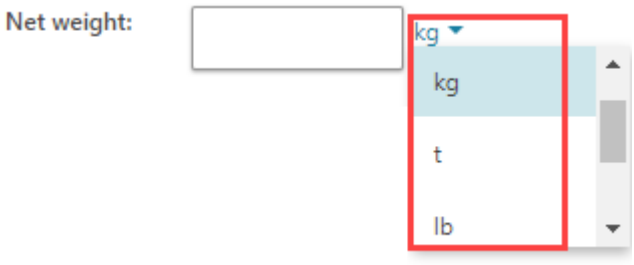
> Tip:
>
> Each property can be overridden in the command line arguments. See the **eclass2json.properties** file for the correct syntax.

Property	Description
targetReleaseVersion	Target release for which the JSON file is intended. This value helps determine which schema is used to create the JSON file.
eclassLevel	Can be either **ADVANCED** or **BASIC**.
status	Specifies the status the data will have in the database after conversion. It can be either **Develop** or **Released**. You can classify in released classes only.
skipDeprecatedDefinition	Does not convert deprecated definitions in the source XML. Siemens Digital Industries Software recommends keeping the default value of **false**.
skipDeprecatedReferences	Does not convert the deprecated references. Siemens Digital Industries Software recommends keeping the default value of **true**.
skipExistingIRDI	Adds the **skipExistingIRDI** option to the JSON file so that existing administrative objects are skipped during import.
addAsNewRelease	Adds the **addAsNewRelease** option to the JSON file so that the administrative objects are added as a new release. This begins a new revision chain in the database allowing the classification hierarchy to **contain multiple releases**. Additionally, you can specify that only objects of a specific type are added as a new release.
objectCountPerFile	Specifies the maximum number of objects described in a file.
topLevelNode	Specifies the IRDI of the root node in the classification hierarchy that contains the segment data that you import. The default value contains a variable for the revision level of the ECLASS release that you are importing. This results in a unique top level node for each new ECLASS release that you install.

Property	Description
topLevelNodeName	Specifies the name of the root node in the classification hierarchy in that contains the segment data that you import. The default value contains a variable for the revision level of the ECLASS release that you are importing. This results in a unique top level node for each new ECLASS release that you install.
WSO.revision_separator	Specifies the separator between item ID and item revision that is used at import when creating a classified item revision in Teamcenter. This property is used for converting BMEcat data only.
WSO.revision_id	Specifies the revision ID which is used to create a revision of this ID (or to classify a revision of this ID) when importing the product data into Teamcenter. This property is used for converting BMEcat data only.
WSO.revision_type	Specifies the workspace object type created when importing the converted JSON files into Teamcenter. This property is used for converting BMEcat data only.
WSO.classifyRevision	Specifies that the item revision is classified during import. If this option is set to **false**, the item is classified. This property is used for converting BMEcat data only.

Import ECLASS units

When viewing classification data, you can convert between units that exist in the database. The ECLASS standard data packages include a large number of unit definitions. These are contained in an XML file in the ECLASS data package that you downloaded. To be able to convert between units when using ECLASS data, you must first import the units into the database.

Units with conversion	Units without conversion
	

1. In a command line, navigate to the directory containing the unit definition file and convert it to JSON with the **eclass2json** utility:

%TC_PERL% %TC_BIN%\eclass2json\eclass2json.pl -objectType:units -inputFile:eClass11_1_UnitsML_EN_DE.xml

2. Import the newly created JSON file found in the **jsonOutputDirectory** folder as follows:

   clsutility -create -unit_definitions -request=*JSON file*

The ECLASS units are now available for use in the user interface. When converting between measures, the value is stored in the object's unit system (the one chosen in the **Unit System** radio buttons at the top of the **Properties** section). For example, if a property's base unit is **mm** and if the unit system of an object is metric, even if you set the value of the **Net weight** property to **lb** or **kg**, internally, the value is converted to **mm** before being stored.

Importing data (datasets) associated with BMEcat property records

When importing ECLASS product data originally in BMEcat file format, there are generally additional files associated with each property record. These must be imported into the database and associated to each imported classification object as a dataset using a specific named reference, depending on the file type. By default, metadata defining the datasets is included in the BMEcat file. For example:

```
"AttributeIDsForDocumentation": {
    "0173-1#AAQ680": {
      "PropertyIDsForDocumentation": {
        "0173-1#AAC311": "FileName",
        "0173-1#AAS193": "DocumentName",
        "0173-1#AAM660": "DocumentDescription"
      }
    }
```

When importing, the **clsutility** uses a configuration file to determine which property IDs relate to the datasets. Additionally, the configuration file specifies the named reference type used to import each file type. This configuration file, **ClsPropertyRecordDatasetConfiguration.json** is found in the *****TC_DATA***classification** directory. The default settings in the file correspond to the common property IDs and dataset types found in BMEcat data, but you can modify this file to suit your needs. The file also defines the types of relations with which the attachments are imported.

```
{

  "AttributeIDsForDocumentation": {
    "0173-1#AAQ680": {
      "PropertyIDsForDocumentation": {
        "0173-1#AAC311": "FileName",
        "0173-1#AAS193": "DocumentName",
        "0173-1#AAM660": "DocumentDescription"
      }
    }
  },
  "ClassifiedObjectAttachment": {
```

```
"DocumentRevision": {
  "JPG": {
    "DatasetType": "JPEG_Thumbnail",
    "NamedReference": "JPEG_Reference",
    "RelationType": "IMAN_manifestation"
  },
  "PNG": {
    "DatasetType": "PNG_Thumbnail",
    "NamedReference": "PNG_Reference",
    "RelationType": "IMAN_manifestation"
  },
  "DXF": {
    "DatasetType": "UGPART",
    "NamedReference": "Autocad-files",
    "RelationType": "IMAN_Rendering"
  },
  "STP": {
    "DatasetType": "UGPART",
    "NamedReference": "STEP-files",
    "RelationType": "IMAN_Rendering"
  },
  "PDF": {
    "DatasetType": "PDF",
    "NamedReference": "PDF_Reference",
    "RelationType": "IMAN_reference"
  },
  "DEFAULT_EXT": {
    "DatasetType": "MISC",
    "NamedReference": "MISC_BINARY",
    "RelationType": "IMAN_reference"
  }
},
"ItemRevision": {
  "JPG": {
    "DatasetType": "JPEG_Thumbnail",
    "NamedReference": "JPEG_Reference",
    "RelationType": "IMAN_manifestation"
  },
  "PNG": {
    "DatasetType": "PNG_Thumbnail",
    "NamedReference": "PNG_Reference",
    "RelationType": "IMAN_manifestation"
  },
  "DXF": {
    "DatasetType": "UGPART",
    "NamedReference": "Autocad-files",
    "RelationType": "IMAN_Rendering"
  },
  "STP": {
```

```
        "DatasetType": "UGPART",
        "NamedReference": "STEP-files",
        "RelationType": "IMAN_Rendering"
      },
      "PDF": {
        "DatasetType": "PDF",
        "NamedReference": "PDF_Reference",
        "RelationType": "IMAN_reference"
      },
      "DEFAULT_EXT": {
        "DatasetType": "MISC",
        "NamedReference": "MISC_BINARY",
        "RelationType": "IMAN_reference"
      }
    }
  }
}
```

Setting up classification with artificial intelligence

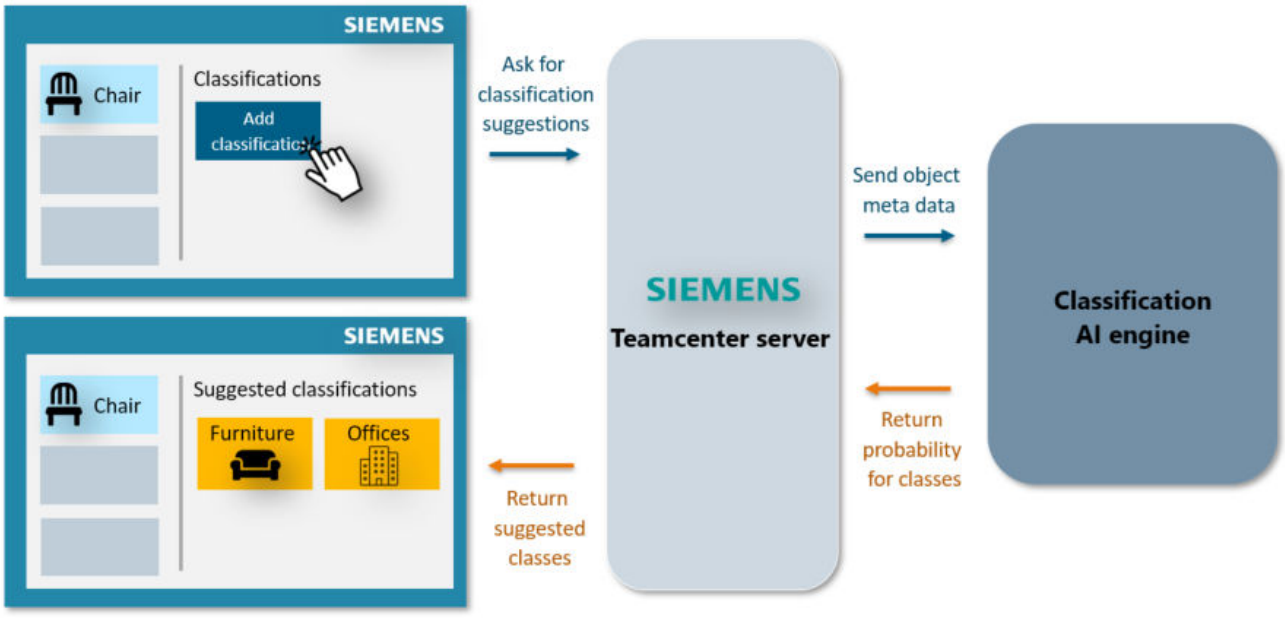
Overview of configuring classification artificial intelligence

> Note:
>
> Ensure that you have the appropriate license to use this feature. For more information, contact customer support.

Using artificial intelligence (AI), Active Workspace can guide you to the correct class in which to classify new objects. This is beneficial if your hierarchy is very large so you do not have to manually navigate to the desired class or remember the required class name. This saves time, especially for workflows where you often classify into similar classes.

The classification AI engine is based on TensorFlow, an open source machine-learning library. It can, optionally, use Geolus, the Teamcenter shape search engine, to recommend a class based on the shape of the object being classified. After being trained on a database, the engine receives object metadata from the Teamcenter server and returns the probabilities for potential classes. You can specify which classes are displayed in the user interface based on these probabilities. For example, only classes with a probability of higher than 50% are displayed as potential classification classes.

To configure classification AI, the AI engine must first be trained on the data in your database. The training outputs a data model (files) that is used to deliver suggested classes.



Training the AI engine is carried out by running a utility. This utility need only be run when necessary. If your data changes often, new training is required. Training can be time-consuming depending on how much data you have.

The general process to install and configure classification AI is as follows:

1. (Optional) To use the Geolus shape search with classification AI, the Geolus shape search engine must be installed and configured on the Teamcenter server.

   For details refer to the Geolus documentation.

2. **Install the feature**.

3. **Modify the required preferences**.

4. **Train the AI engine**.

5. View the suggestions in the Active Workspace client.

> Note:
>
> If you use both traditional and classification standard taxonomy (CST) data, the training is performed on CST data only.

Install classification AI components

1. Ensure that you have **installed Classification Active Workspace application**.

The installation creates a new folder in the Teamcenter root directory named **classification**. This folder contains all the files for classification AI.

> Note:
>
> During the installation, the URL of the AI microservice is displayed. Note this URL as it is needed in the next step.

2. Add the URL of the microservice to the **TC_Microservice_Base_URL** preference.

   This URL is generally:

   http://*machine-name-or-IP-address-of-where-microservices-are-installed***:9090**

Deploying the classificationaiserving Docker image on Linux

Prior to deploying the image, you must perform the steps in *Install classification AI components*.

1. Deploy the image to the appropriate stack by navigating to the machine and directory where microservices are installed. The *Microservices-home-directory***/container** directory contains the Docker image and corresponding YML file.

   Run the following command:

   docker stack deploy --compose-file classificationaiserving.yml *your-stack*

   The service is created and added to the stack.

2. Test that the service is running by typing the following in the address field of a browser:

   *machine-or-IP-address-running-microservices***:9090/cais**

For more information about general Docker administration commands, see their website:

https://docs.docker.com/engine/reference/commandline/stack/

Train the classification AI engine

Training the artificial intelligence engine on your data involves running a script that extracts data from the database in a format that the engine can subsequently use to create the training model. The properties that are extracted and trained on are specified in the **CLS_AI_Object_Properties** preference.

1. Set the required preferences (see additional preferences **here**):

   CLS_AI_Enable_AI_Engine

Set this preference to **true** to enable class suggestions.

TC_Microservice_Base_URL

Specifies the URL of the microservice that supports classification AI. This URL must be set during installation of the feature and has this format:

IP of machine on which microservices are installed:port

A common port used is **9090**.

To enable Geolus with classification AI, additionally set this preference:

CLS_AI_Enable_Geolus

Enables the additional comparison of shape parameters using the Geolus search engine with classification AI.

2. In a Teamcenter command prompt window, navigate to the *TC_ROOT***classification\ai\bin** directory and run the following script:

   runClsAITraining

   This script trains the AI engine on the data in your database.

   - The script creates two CSV files, **meta_data_train.csv** and **meta_data_val.csv**. The **meta_data_train.csv** contains the data used to train the model. The **meta_data_val.csv** file contains a subset of this data that is used to validate the training model. When complete, the model named **cait-artifacts-TrainOutput_***timestamp*, is uploaded to the file repository and the two files used to generate it are subsequently deleted from your local directory.

     Tip:

     If there are problems with the configuration, you can **run a utility to generate these files and retain a local copy**.

   - To obtain assistance for the **runClsAITraining** script, run it including the **-h** argument.

   Each time you run the script, you create a new training model that has no memory of the last run.

   Tip:

   As the training is run using a batch script, you can schedule it to run at regular intervals.

3. (Optional) **Set remaining classification AI preferences**.

Using classification AI with traditional and classification standard taxonomy data

If you use both traditional and next generation classification data, internally, different training engines are used to prepare the class predictions. To ensure that AI is correctly set up for using classification standard taxonomy data, do the following:

1. Set the **CLS_is_presentation_hierarchy_active** preference to **true**.

2. Run the training. Pay attention to the message displayed when you do this. The message should read **Searching database for Next Generation Classification Objects**, and not **Searching database for Traditional Classification Objects** as is displayed when training traditional data.

   A model specific to next generation or classification standard taxonomy data is trained and uploaded to the file repository. If AI is set up correctly, the serving component immediately picks up the new model.

Only users that have presentation layer (Next Generation Classification) enabled should use the model trained with the **CLS_is_presentation_hierarchy_active** preference set to **true**. Training may not give accurate suggestion probabilities for traditional Classification users who are getting predictions from a Next Generation AI model, and vice versa.

Classifying objects in batch mode

A classification administrator can run a utility that classifies multiple objects simultaneously based on definable properties. This classification process is carried out using artificial intelligence. The process sends the objects to a workflow. Specified users receive a notification and can then decide whether to accept the suggested classification or, if it is not appropriate, delete it.

> Note:
>
> Batch classification is not available for traditional basic classification.

1. Set the following preferences:

   CLS_AI_Object_Properties

   ICS_classifiable_types

   CLS_AI_Engine_Probability_Cutoff

   CLS_AI_Engine_URL

2. Run the **cls_ai_auto_classify** utility in dry run mode (without specifying the **-classifyObjects** argument), and examine the results found in the **AutoClassifyReport.csv** file residing in the **-outputPath** location that you specify.

The CSV file lists the objects that are found based on the preference settings, the probability that they match the suggested class, and whether they will be classified by the utility.

3. If you are satisfied with the results from the dry run, re-run the **cls_ai_auto_classify** utility and specify **-classifyObjects**.

   The objects found by the utility are sent to the **Classification AI Review Classifications** workflow.

4. Select the classification experts responsible for approving the classifications.

   The reviewers receive a notification and can approve the classification for the object, thereby classifying it, or remove the suggested classification.

   > Note:
   >
   > You cannot delegate the review task to another user.

Troubleshooting classification AI

- **How do I know whether the classification AI server is running?**

  To ascertain whether the classification AI server is running, open a Web browser and type the URL where microservices is installed:

  *IP of machine on which microservices are installed:port***/cais/**

  A common port used is **9090**.

  If the server is running correctly, the following is displayed:

  Connected to Classification AI Serving

- **How do I prevent the training files from being uploaded to the file repo if problems arise during the configuration of AI?**

  The runClsAITraining script calls several utilities:

cls_AI_data_generation	Generates the two CSV files used to create the model and stores them in the following directory: *TC_ROOT***classification\ai\working\cai-data**
app.js	Creates the model from the two CSV files, uploads it to the file repository, and deletes the two CSV files from the local directory.

  If there are problems with configuring AI, run the **cls_AI_data_generation** utility manually to generate the two CSV files but prevent them from being deleted after uploading to the file repository.

The files can then be found in the *TC_ROOT***classification\ai\working\cai-data** directory. This directory contains time-stamped subdirectories. The two relevant training files can be found in the most recent of the subdirectories. You can delete these files after you finish troubleshooting.

- **How do I make sure the data on which the AI is trained is valid?**

  Classification AI trains on the metadata from existing classified objects. To discover the contents of this metadata, you can run an ITK utility, **cls_AI_data_generation**, that lists the existing metadata in a CSV file found in the following directory:

  *TC_ROOT***classification\ai\working\cai-data***time-stamped-folder***

  Run the utility and examine the output to ensure that the data on which it was trained is both meaningful and diverse enough that a pattern can be extracted by the learning algorithm.

- **Where do I find log files?**

  The syslog created during training is stored in your local *%TEMP%* directory. It has the following format:

  classificationaiserving*instance_number***@***node_id***-msf.log**

  For example:

  classificationaiserving1@22504-msf.log

- **Why am I not seeing class suggestions?**

  Ensure that both the **following preferences are set**:

  - **CLS_AI_Enable_AI_Engine**

    If this preference is not set to **true**, Teamcenter will not call the classification AI serving component for suggestions.

  - **TC_Microservices_Base_URL**

    This preference must be set in the URL of the microservices.

- **What preferences are set during the Geolus installation?**

  If Geolus is enabled, the following preferences are set by the installation and can be checked for accuracy:

  SS1_DASS_cert_name

  Defines the full name of the Geolus server certification file.

SS1_DASS_cert_path

Defines the path to the Geolus server certification file.

SS1_DASS_ssl_verify

Defines whether CURL verifies the certificate authenticity of the Geolus server.

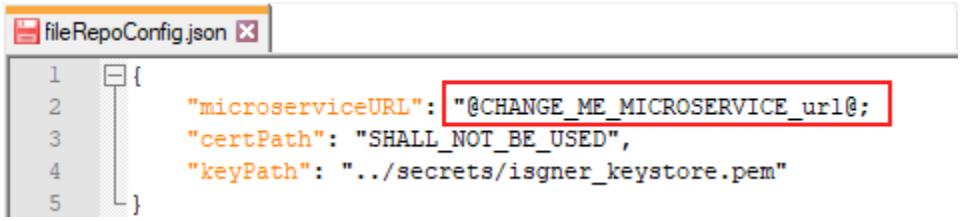
GeolusServer

Defines a URL that points to the Geolus server.

- **What do I need to know about upgrading from an earlier version of AI?**

  When upgrading from a later release, the old training model is still functional and is migrated to the file repository. When you train again, the model is no longer stored locally in the shared directory but automatically uploaded to the file repository. At this point, you can remove the shared directory as well as the old **CLS_AI_Shared_Directory** and **CLS_AI_Engine_URL** preferences.

  After upgrading from version 5.0, open the **aw/microservices/classification_ai_serving_service-1.0.0/config/fileRepoConfig.json** file and change the value of the **microserviceURL** option to the value of the **TC_Microservice_Base_URL** preference.



Troubleshooting the installation and configuration

- **All classification options are installed and indexing is configured but new objects are still not included in search results**

  To use traditional basic classification in the Active Workspace client, it is not necessary to install the presentation layer (**Next Generation Classification**). If, however, you inadvertently do install this option, you *must* also extend the data using **clsutility** to make use of full functionality.

- **Filter categories are localized but facet values are not**

  If you mark any classification property as localizable, you must run the **bmide_modeltool** utility.

- **Search is incorrectly configured**

  The following situations may arise if the search is not correctly configured:

  - You cannot search by classification properties.

- The classification search works but facets are not visible (or there are unassigned facets).
- You cannot search the library elements by classification property.

Correctly configure the search.

- **The incorrect filters are displayed for library elements**

  1. Set the dynamic prioritization preferences to display all the configured filters. This allows you to investigate further.
     - Set **AWC_dynamicPriority_hideSingleValueFacetCategories** to **true**.
     - Set **AWC_dynamicPriority_hideUnassignedValueFacetCategories** to **true**.
  2. Make properties searchable in the global search.

- **The classification search works, but there only a limited number of properties available in the search filters.**

  Increase the number of search facets displayed using the **AWC_classification_facets_threshold** preference. The default value is **200** and the facets are sorted with the most common facets at the top.

  > Caution:
  >
  > Setting this value to a high number can cause a decrease in performance or even cause problems with Solr.

- **Performing a bulk import of classification data**

  Ensure that the **CLS_auto_sync_node_hierarchy** preference is set to **false** before performing a PLM XML import of hierarchy data.

- **What business constants are involved in classification installation?**

  The following business object constants are set in Business Modeler IDE:

  - The **Awp0SearchClassifySearchEnabled** business object type constant is set to **true** at the workspace object level. This turns on the classification search.
  - The **Awp0SearchIsClassifyDataIndexed** business object type constant is set to **true** at the item and item revision level.

    This turns on classification indexing. The object type on which this constant is set is indexed for searching. All subtypes inherit the setting.

- The **Awp0SearchIsIndexed** business object type constant is set to **true** at the item and item revision level.

  The object type on which it is set is indexed for searching. All subtypes inherit the setting.

  > Note:
  >
  > Do not set this constant at the workspace level since this prevents your database from being indexable.
  >
  > Instead, set it true for each object that you want to index. For example, item and item revision.

Verify that these constants are correctly set using the **gettypeconstantvalue** utility, for example:

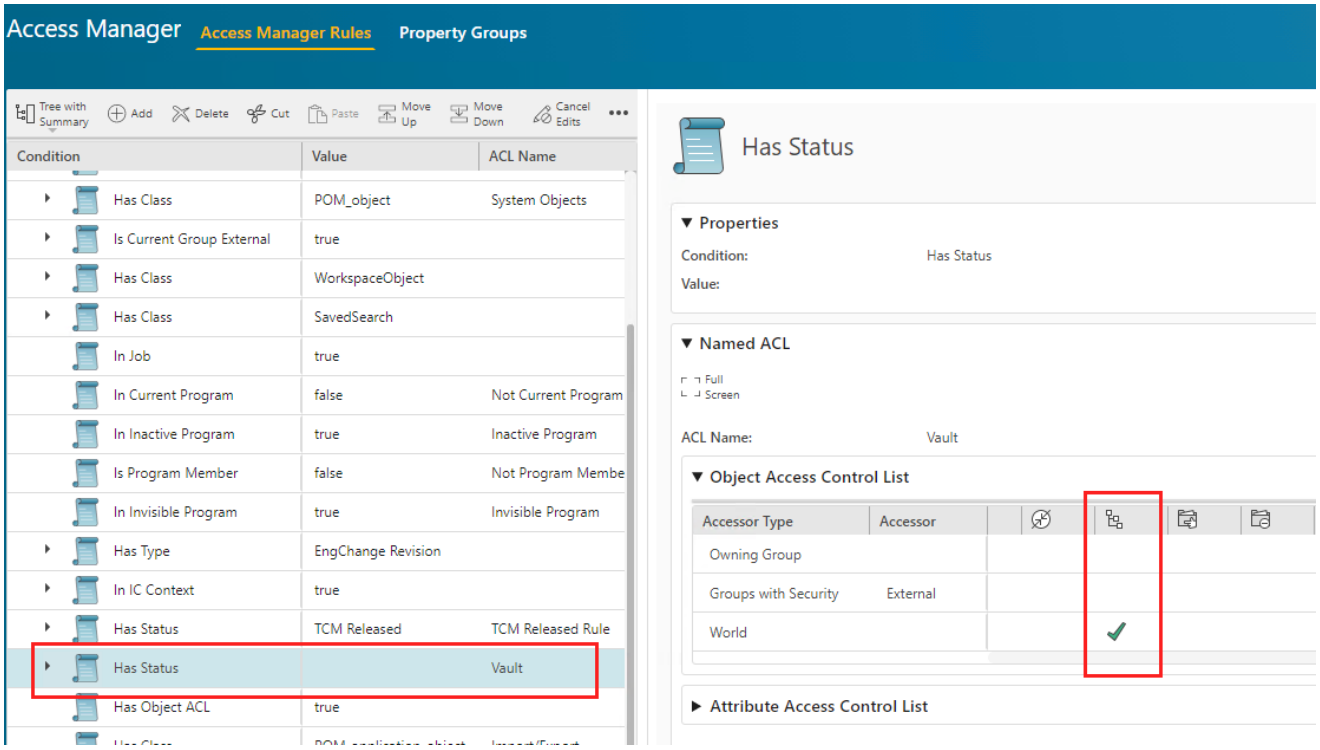
```
gettypeconstantvalue -u=username -p=password -g=dba
-typename=WorkspaceObject
-constantname=Awp0SearchClassifySearchEnabled
```

Give access to users for classifying objects

The following points determine whether you can classify an object (that is, whether the **Classify** button is visible for a selected object):

- The workspace object's type must be listed in the **ICS_classifiable_types** preference. If your business use case requires classifying items instead of item revisions, you must remove the **ItemRevision** entry from this preference.

- You must have write access to the workspace object. If this object is already released, you must have write access to the classification object (**write_ico** access). The workspace object access is set in the **Access Manager** application in rich client. The classification entries in the **Has Status() → Vault** rule must be set to write access.



- You must **set additional classification preferences** depending on the classification features you use.

Configuring classification enforcement

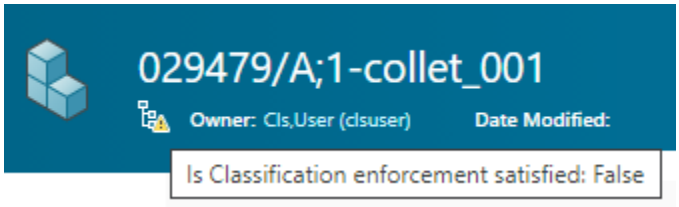
If your business process requires that all objects of a specific type, for example **Part** or **PartRevision**, must be classified, you can enforce classification.

To enforce classification:

- List the internal type name in the **CLS_enforce_classification_on_types** preference.

> Example:
>
> **Part** or **PartRevision**.

When an object of this type is created, it is marked with a warning sign indicating that it has not yet been classified and a tooltip notifying that the enforcement requirement is not fulfilled.



To ensure that classification of these objects occurs, a user can submit the object to a workflow that uses the **Review Classifications** template and assign it to the responsible subject matter expert. The object to be classified is displayed as a target of this workflow and can be opened from within the workflow. After classification, the subject matter expert completes the workflow affirming that the object is correctly classified and the warning sign is removed from the classified object.

Granting access to non-dba users for Classification Manager tasks

While configuring Access Manager rule conditions for a classification user, any user can be a classification admin user. It is not necessary for the user to be defined as a dba user for performing classification administrator tasks.

You must create an ACL to grant the **Read**, **Write**, **Delete**, **Change**, and **Copy** access for any non-dba user to access the Classification Manager application and perform classification administrator tasks.

8. Administering classification

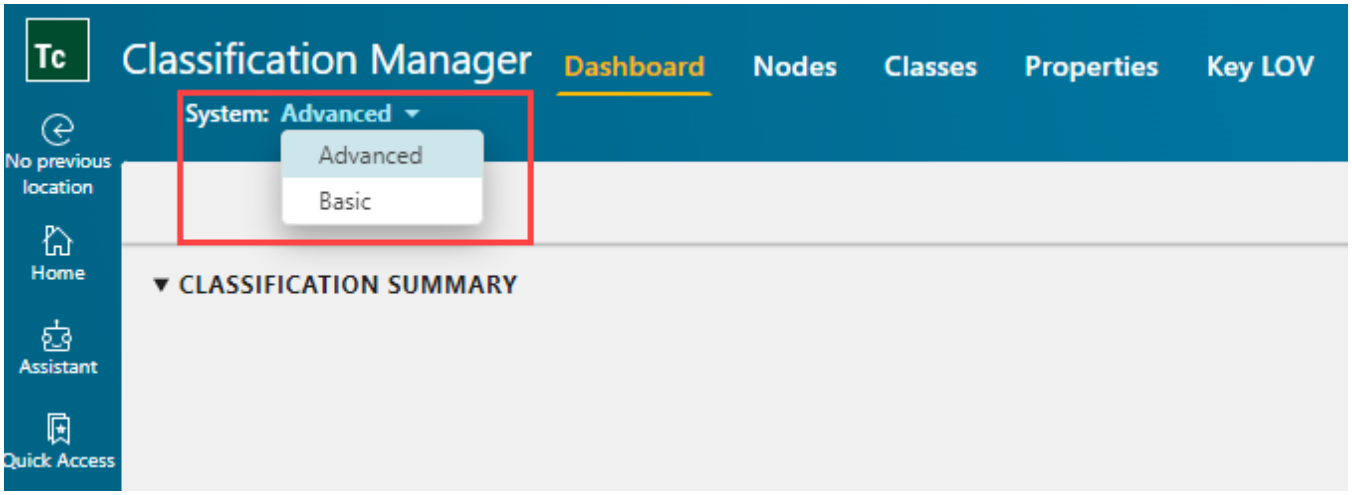
Viewing your hierarchy definitions

The basic and advanced classification hierarchy is created of nodes, classes, properties, and key-LOV administrative objects. These definitions are displayed in **Classification Manager**. The **Classification Manager** provides an overview of every classification definition in the database on the **Dashboard** tab, as well as details about each object on the **Nodes**, **Classes**, **Properties**, and **Key LOV** tabs.

The **CLASSIFICATION MANAGER** tile is found in the **Active Admin** workspace.

By default, the **Active Admin** workspace is mapped to the **dba** group and role. You can add other groups and roles as needed. Alternatively, you can create your own workspace that includes the **CLASSIFICATION MANAGER** tile.

You can view the classification definitions for basic or advanced classification by selecting either the **Basic** or the **Advanced** system in **Classification Manager**.



Use the **Import** button on any of the tabs in **Classification Manager** to import administrative objects (key-LOV definitions, property definitions, class definitions and node definitions). The objects must

always be imported in the correct order. For advanced classification you can import JSON, OntoML, or BMEcat file format.

Authoring hierarchy definitions

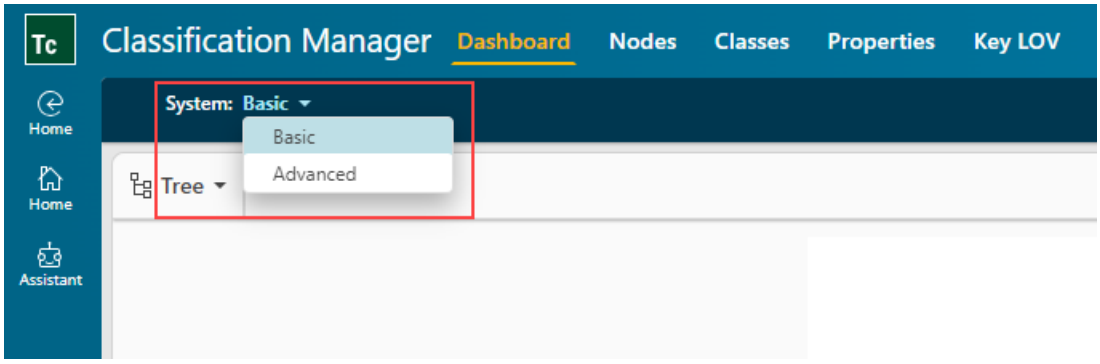
Creating classification hierarchy definitions

You use Classification Manager to define the nodes, classes, properties, and Key LOVs that form the classification hierarchy for basic and advanced classification. You also define and format the attributes that, when associated with a class, determine the type of information that is stored. The order when creating a classification hierarchy is to first create the Key LOV definitions, then the property definitions, then the class definitions, and finally the node definition so that the node is visible in the classification hierarchy for end users.

Use Classification Manager to do the following basic tasks:

- Create a Key LOV definition
- Create a property
- Create a class
- Create a node

You can create the classification definitions for basic and advanced classification. Select either **Basic** or **Advanced** from the **System** list available on the **Classification Manager** dashboard before creating the classification definitions.



Creating Key LOVs

Overview of Key LOVs

In Classification, Key LOVs are lists of values, such as a list of countries.

You can do the following using Key LOVs:

- Define values for the Key LOV

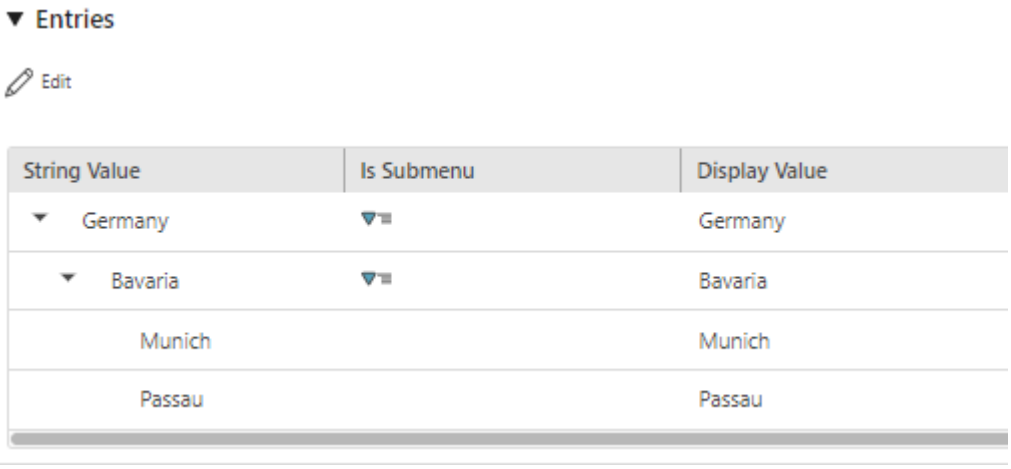
  In Classification, values for Key LOVs are called Entries.

- Define submenus for Key LOV entries

  Submenus are child values that you add to the Key LOV entries, which creates cascading Key LOVs.

  Example:

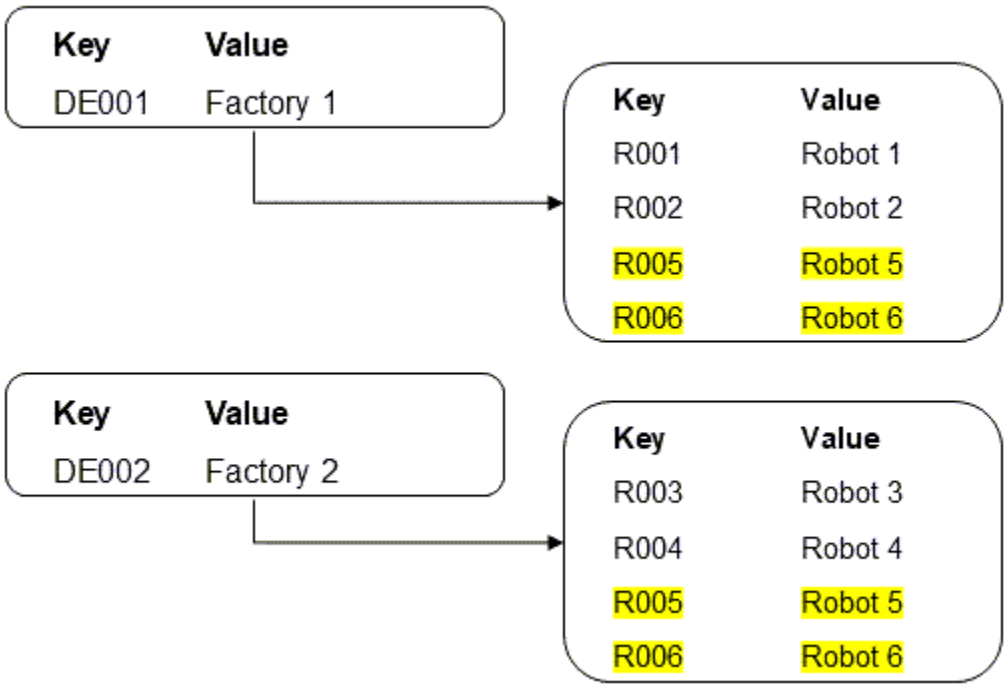
  For a Key LOV that has an entry for the country, you can create submenus for the city and state.

- Reuse entries in a Key LOV

  When classifying data, you may want to assign the same values to different objects. For example, if the same type of robot is used across multiple manufacturing sites, reusable key-value pairs help you define the robot data only once and then assign the robots to different factories.

  In the following example, Robot 5 and Robot 6 are reused in Factory 1 and Factory 2.



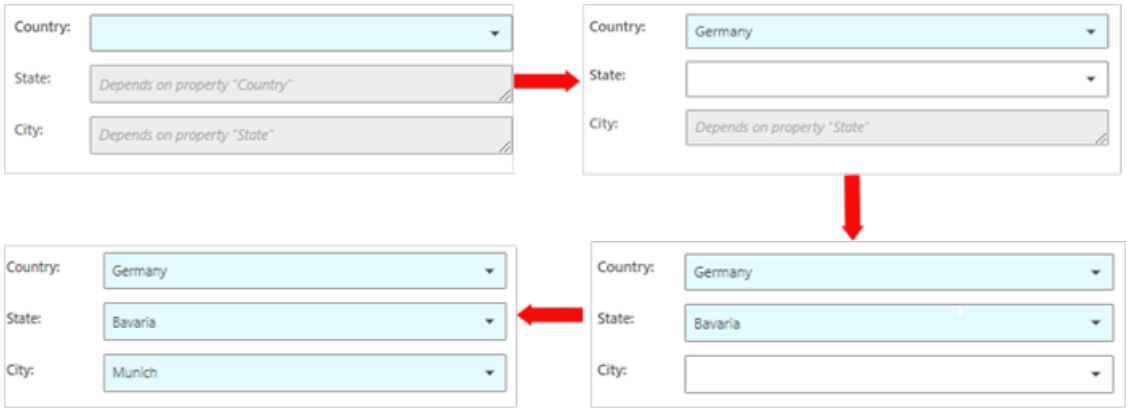
Reusable key-value pairs help you:

- Define a Key LOV once and use it multiple times across different classes or contexts.
- Make changes everywhere that an entry is used when you update key-value pairs.

- Dependent Key LOVs

  In a dependent Key LOV, setting the value of one Key LOV makes the next dependent value available for selection.

  In the following example, in a dependent Key LOV, if you select the country, the dependent Key LOV is available for selection. Additionally, when you select the state, the dependent Key LOV is available for selection.



- Dynamic LOVs

  In Business Modeler IDE (BMIDE), you can create Dynamic LOVs. Dynamic LOVs are lists of values where the values are shown at runtime by querying the database, such as a list of users or a list of parts belonging to a particular group.

  You can use these dynamic LOVs in Classification by associating the dynamic LOV with a Key LOV.

Create a Key LOV definition

You use Key LOVs to define one or more values that can be set for specific classification attributes. Once created, these lists are stored in the data dictionary and can be used and reused as required.

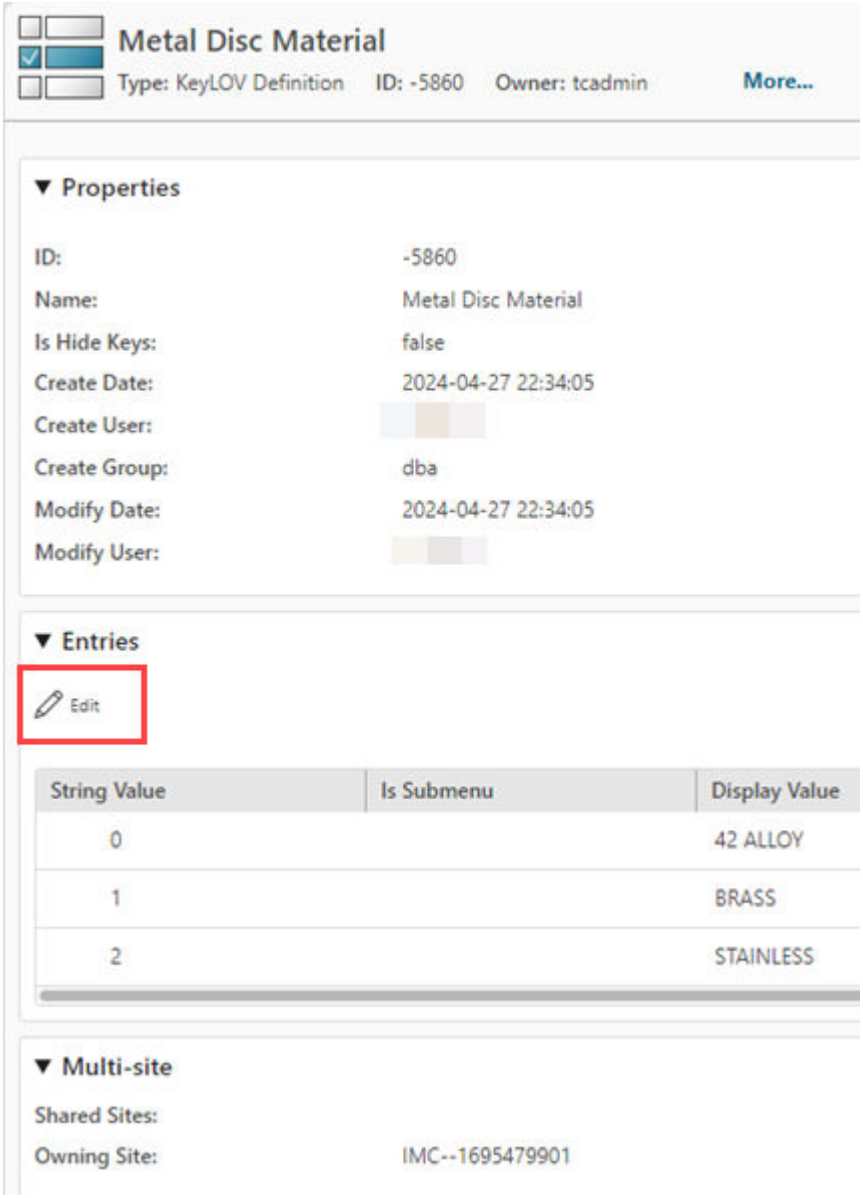
Procedure

1. On the **Classification Manager** dashboard, click the **Key LOV** tab.
2. Click **New** ⊕.
3. In the **Add Key LOV** panel, specify the following details:
   - **Namespace**
   - **ID**
   - **Revision**
   - **Name**
   - **LOV Items**
4. Click **Add**.

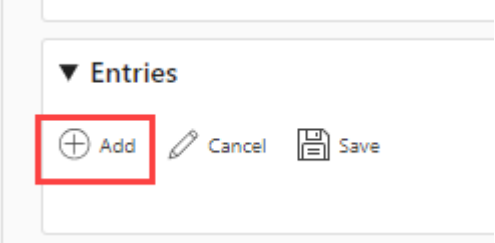
The Key LOV definition is created.

5. After creating a Key LOV, you can add the entry for the Key LOV definition.

6. Open the Key LOV definition. Under **Entries**, click **Edit** in order to add entries to the Key LOV or to edit previously added entries.



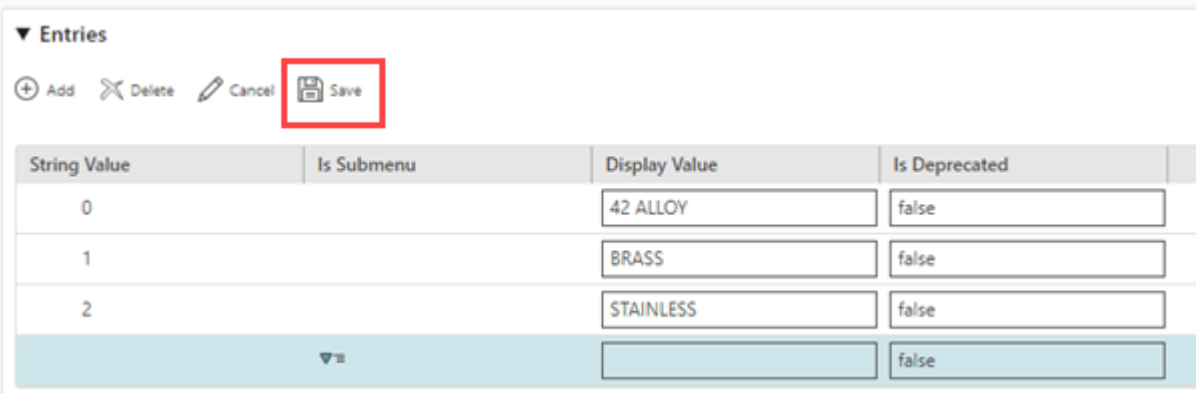
7. Click **Add** .



8. Specify the details as required and click **Add**

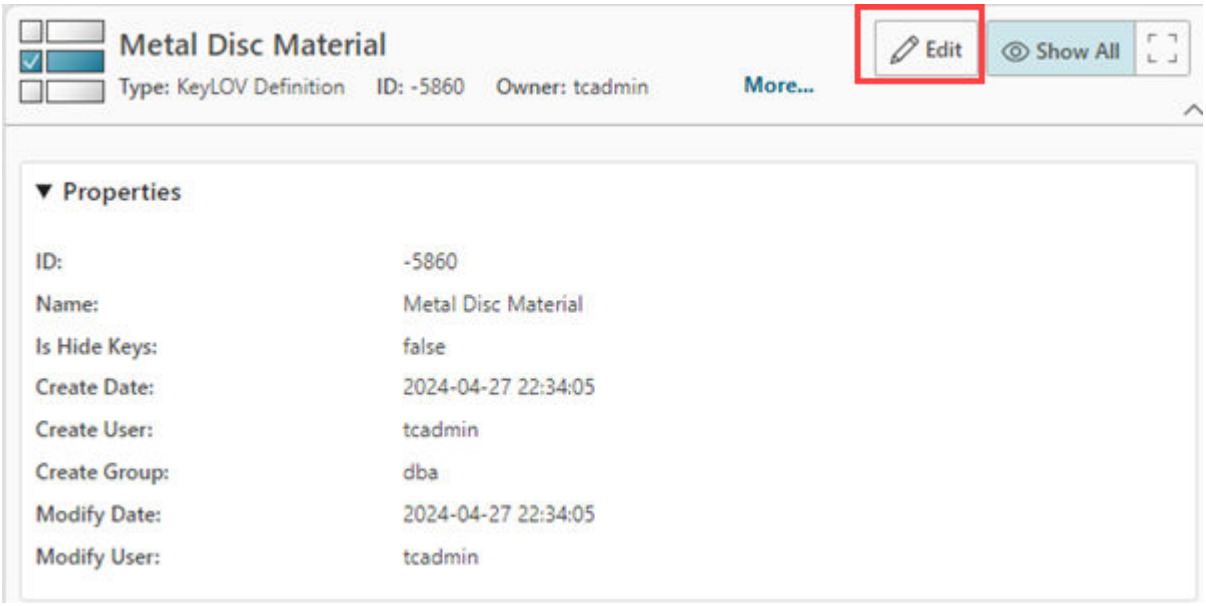
   You can add more entries to the Key LOV as required.

9. After all the entries are added, click **Save**.



10. To edit Key LOV definition, click **Edit**.

    The editable cells are activated.



11. Enter or update the Key LOV definition as required.

12. Save your changes for the Key LOV definition, by clicking **Save**. To cancel your edits, click **Cancel**.

Add submenus to a Key LOV

You can add submenu entries to the Key LOV entries, or you can add submenu entries to other submenus, creating cascading Key LOVs.

These steps show how to create state and city submenus for the country Key LOV entry.

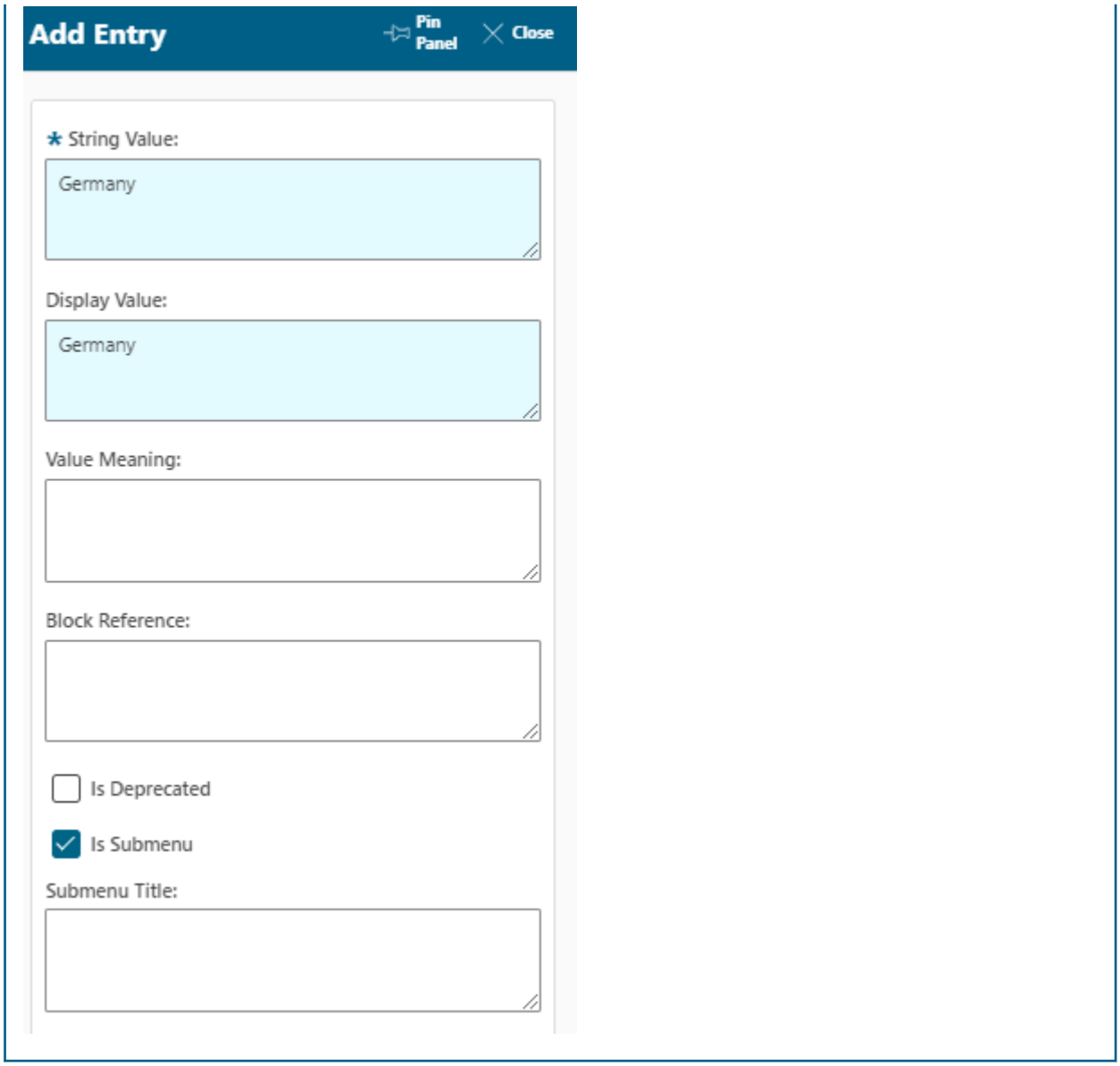
Procedure

1. On the **Classification Manager** dashboard, click **Key LOV**, and open or create a Key LOV.

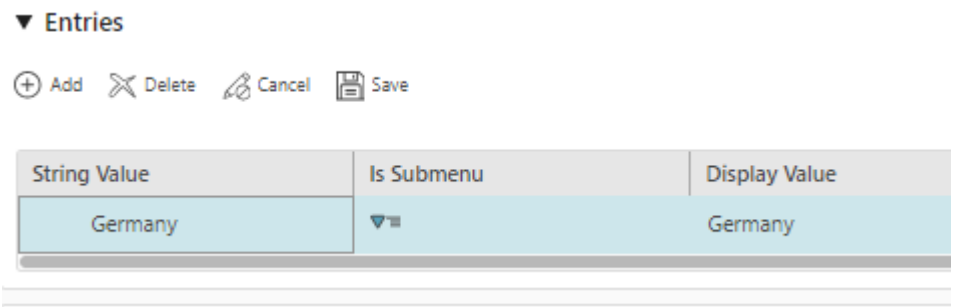
2. Create an entry for the Key LOV and in the **Add Entry** panel, select **Is Submenu** and click **Add**.

   Example:

   Create an entry for the country, such as Germany.



3. To add a submenu entry to a Key LOV entry, select the Key LOV entry and click **Add**.

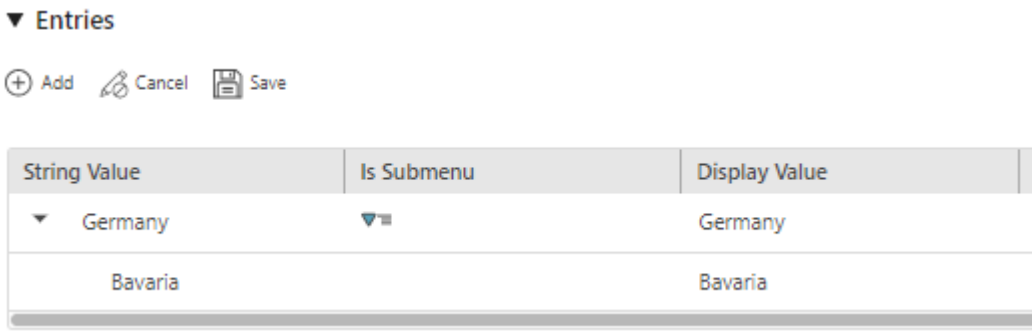


4. In the **Add Entry** panel, add details about the submenu entry and click **Add**.

Example:

Add the values for the state, such as Bavaria.

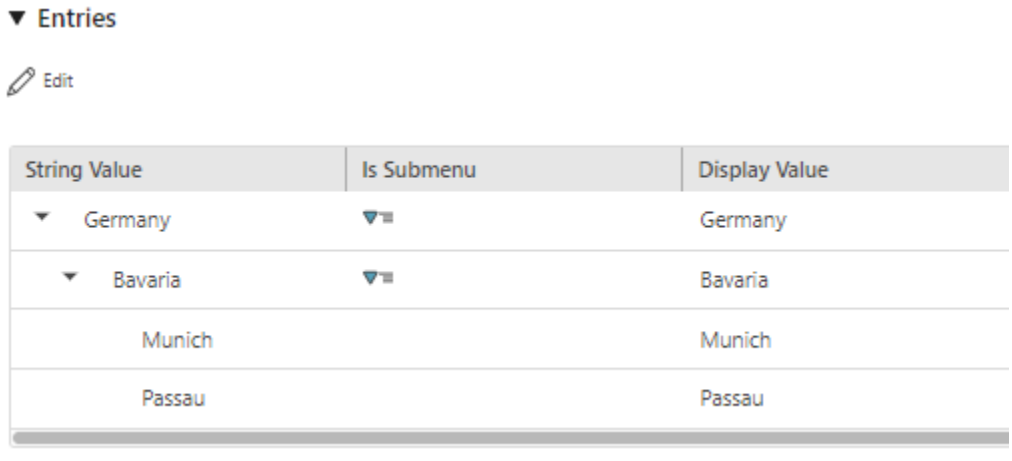
The submenu entry is added under the Key LOV entry.



5. You can also add submenus to existing submenus by selecting the **Is Submenu** option in the **Add Entry** panel.

6. In the **Entries** section, click **Save**.

Results



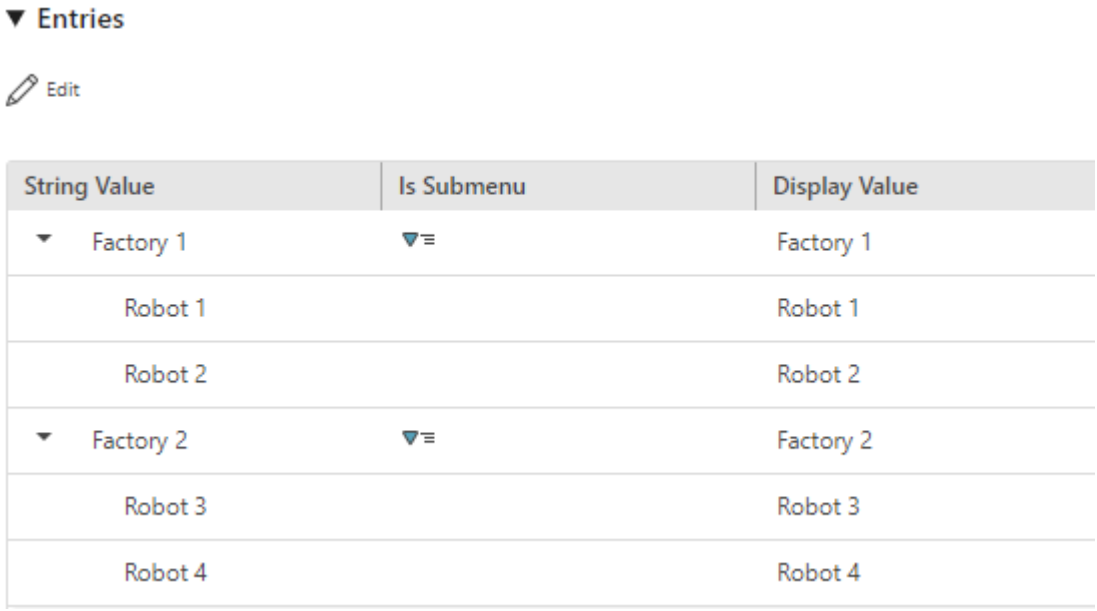
Reuse entries in a Key LOV

When classifying data, you may want to assign the same data to different objects. For example, if the same type of robot is used across multiple manufacturing sites, reusable key-value pairs help you define the robot data only once and then assign the same robots to different factories.

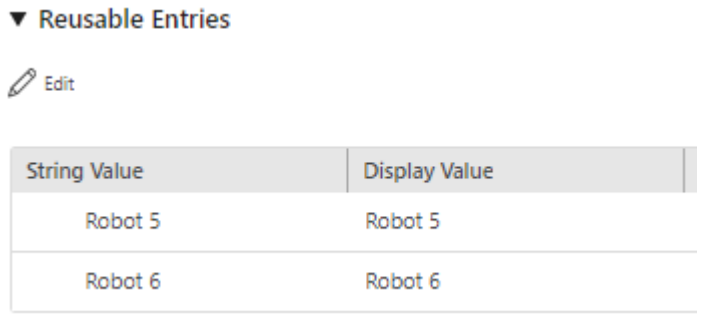
Procedure

1. On the **Classification Manager** dashboard, click **Key LOV**.
2. Open a Key LOV and click **Edit.**

   Add submenus to the Key LOV.



3. Click **Edit** in the **Reusable Entires** section, and then click **Add** to add reusable Key LOV entries.
4. Add the reusable Key LOV entries as needed.

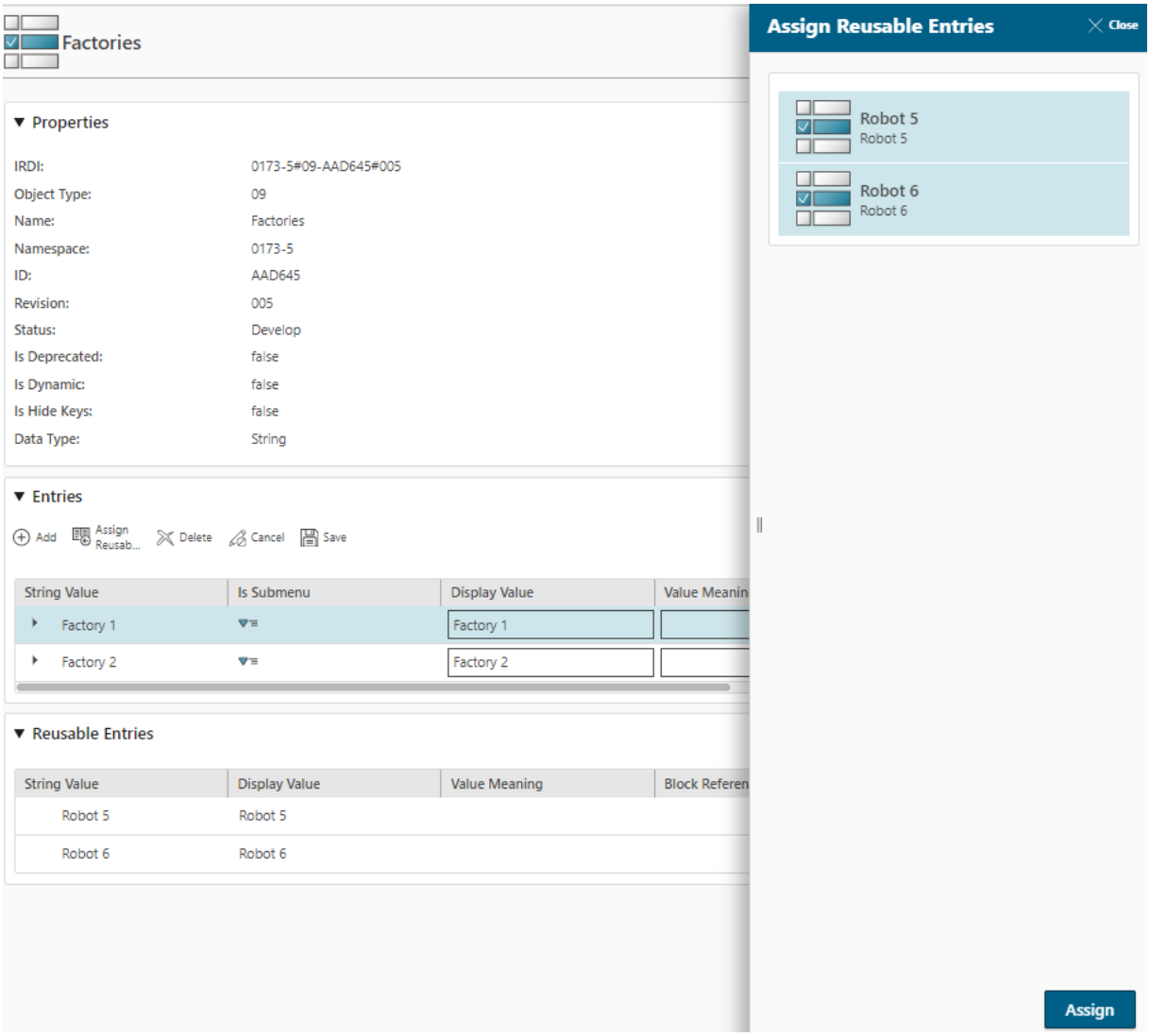


5. To reuse the entries, add the reusable entries in the **Entries** section as follows:

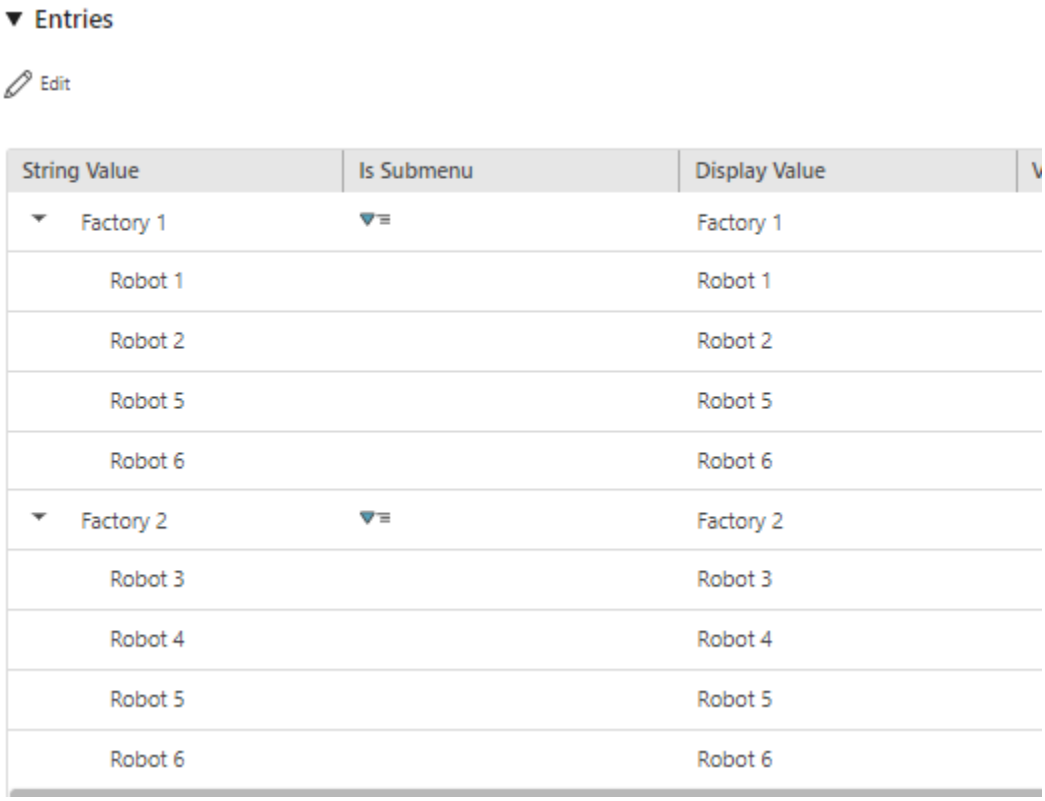
   a. Click **Edit** in the **Entries** section.

   b. Select the entry where you want to add the reusable Key LOV values and click **Assign Reusable Entries**.

c. In the **Assign Reusable Entries** panel, select the reusable entries you want and click **Assign**.



The reusable Key LOV entries are added in the **Entries** section.



Create a dependent Key LOV

In a dependent Key LOV, setting the value of one Key LOV makes the next dependent value available for selection.

The process for creating an dependent Key LOV is as follows:

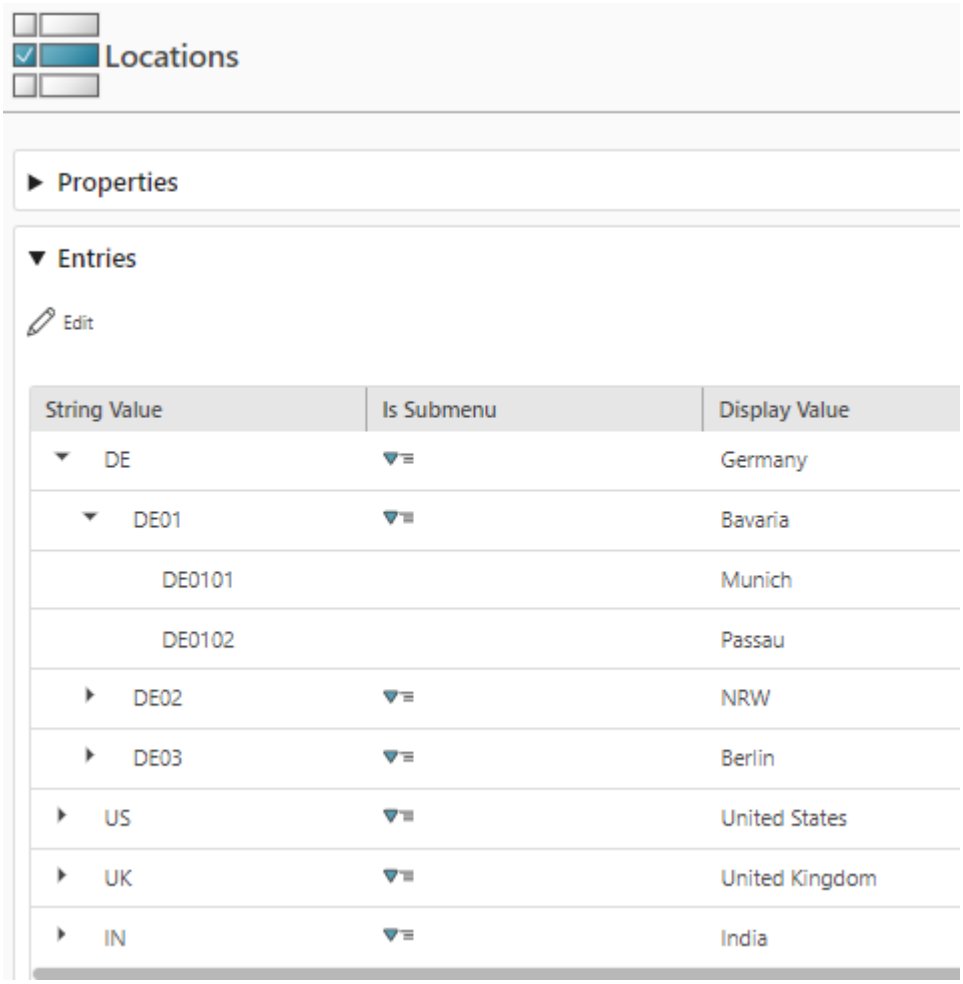
- Create a Key LOV and add the entries and sub menu entries such as entries for country, state, and city.
- Assign the Key LOV to the respective properties. For example, assign the Key LOV country to the country property, the state Key LOV to the state property and so on.
- Assign the properties to the respective class.
- Update the class attributes with information about the level of the dependency and the dependent attribute.

Procedure

1. On the **Classification Manager** dashboard, click **Key LOV**.
2. Create entries and submenus for the Key LOV.

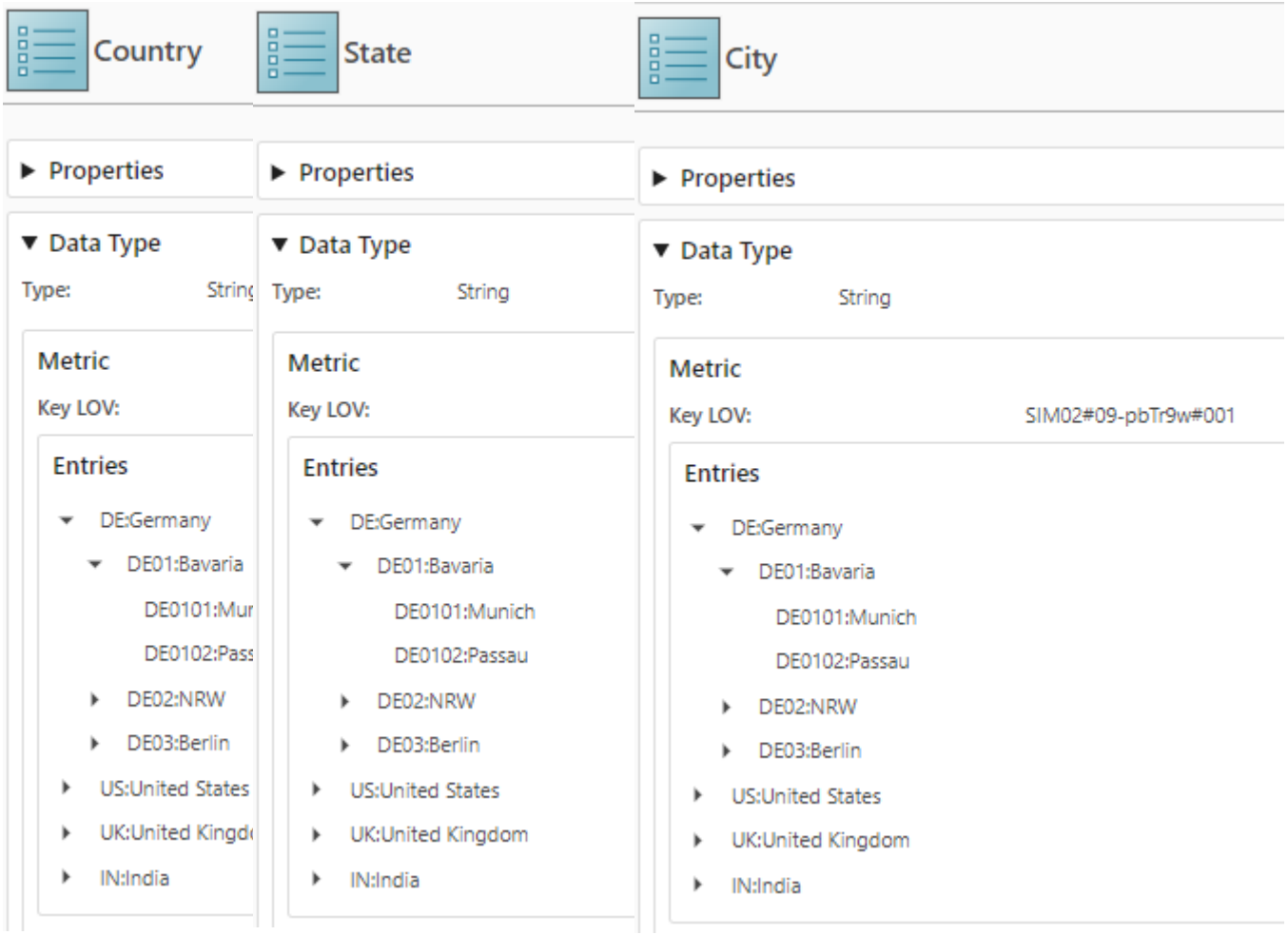
Example:

Create a Key LOV with entries for Country, State, and City.

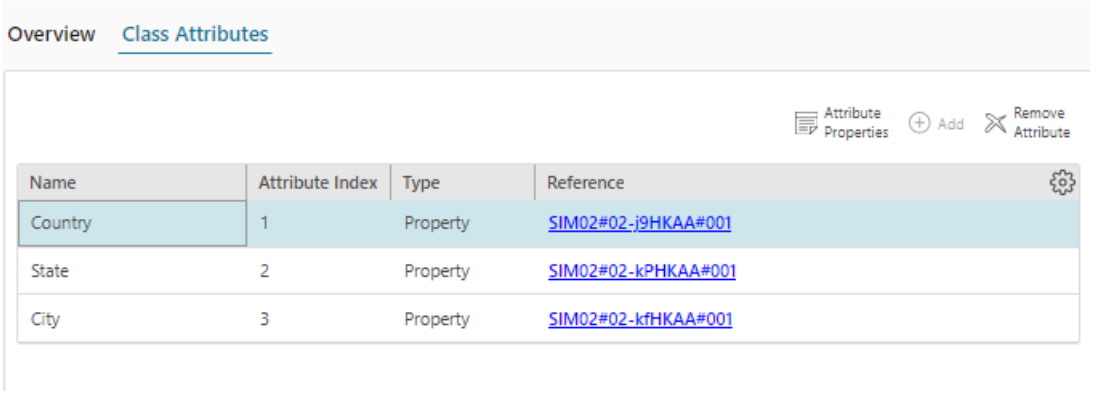


3. Assign the Key LOV you created to the respective properties.

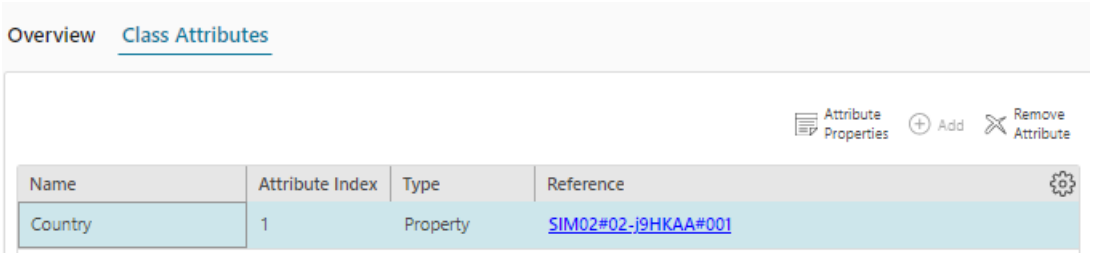
   For example, create properties for Country, State, and City, and assign the Key LOV to these properties.



4. Assign the properties to an application class.

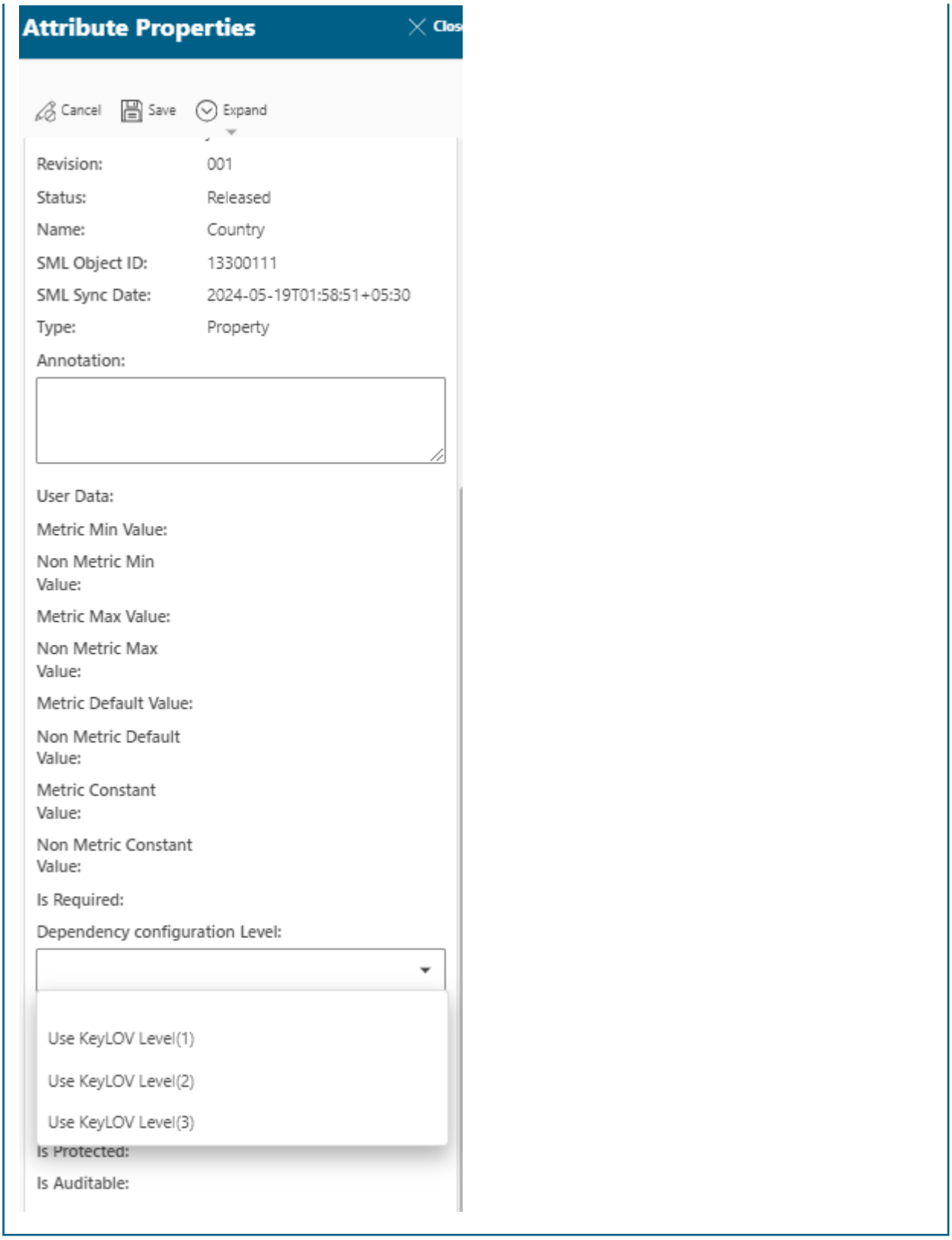


5. To assign the hierarchy level of the Key LOV, select a class attribute and click **Attribute Properties** .



6. In the **Attribute Properties** panel, click **Edit** and select the hierarchy level from the **Dependency configuration level** list.

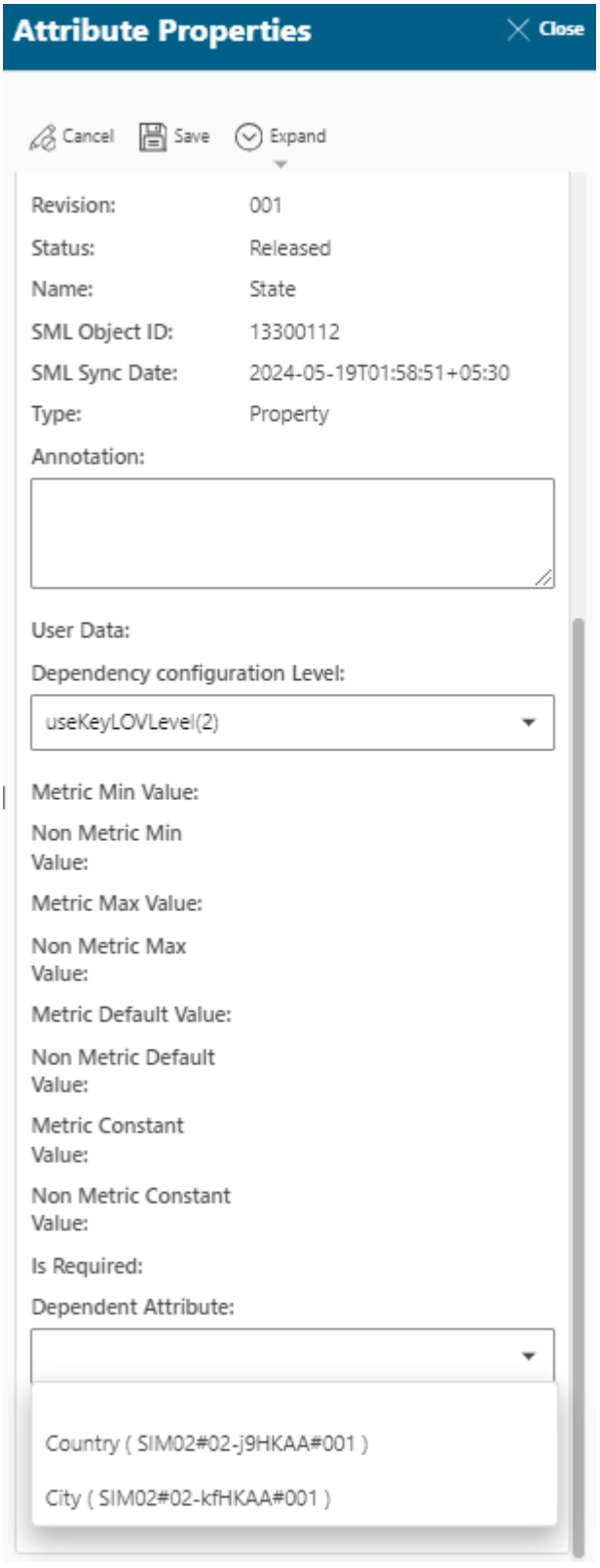
> Example:
>
> Because Country is at the top of the hierarchy, select **Use KeyLOV Level(1)** for Country, select **Use KeyLOV Level(2)** for State, and select **Use KeyLOV Level(3)** for City.



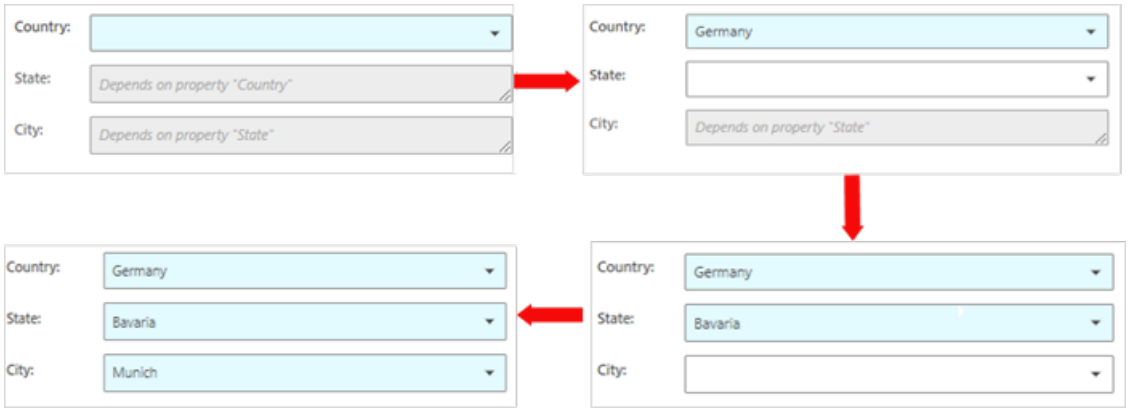
7. To specify the dependency between two attributes, select a class attribute and click **Attribute Properties**.

8. In the **Attribute Properties** panel, click **Edit**, and then select the dependent attribute from the **Dependent Attribute** list.

> Example:
>
> State is dependent on Country. Therefore, for State, select the dependent attribute as Country. For City, select the dependent attribute as State.



You can use the interdependent Key LOV in classification after releasing the class. If you select Country, the dependent Key LOV is available for selection. Also, when you select State, the dependent Key LOV is available for selection.



Using dynamic LOVs created in BMIDE

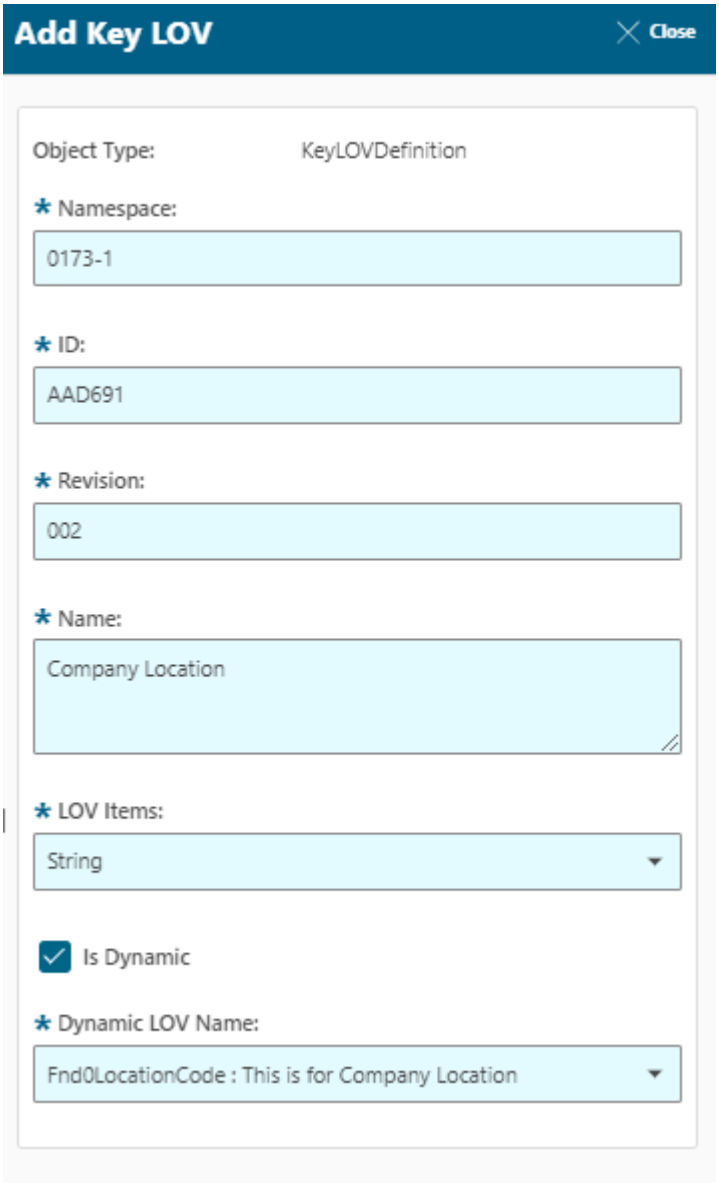
Dynamic LOVs are lists of values where the values are shown at runtime by querying the database. For example, a list of users or a list of parts belonging to a particular group. You can use these dynamic LOVs in Classification by associating the dynamic LOV with a Key LOV.

You can create Dynamic LOVs in Business Modeler IDE (BMIDE).

Procedure

1. On the **Classification Manager** dashboard, click **Key LOV** tab.
2. Click **New** ⊕.
3. In the **Add Key LOV** panel, specify the following details:
   - **Namespace**
   - **ID**
   - **Revision**
   - **Name**
   - **LOV Items**

     String, integer, and double data types are supported.
4. From the **Dynamic LOV Name** list, select a dynamic LOV.



5. Click **Add**.

Create a property

A *property* is used to describe the attributes of a class. Attributes can be of any type, including string, integer, double, boolean, and reference.

Procedure

1. On the **Classification Manager** dashboard, click the **Properties** tab.

2. Click **New** ⊕.

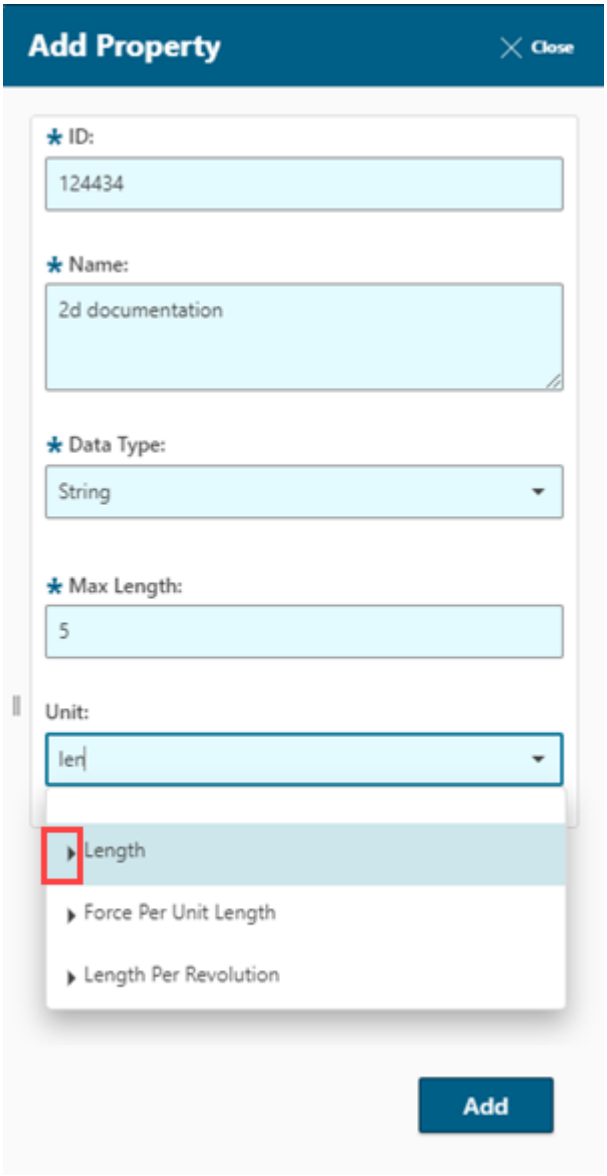
3. In the **Add Property** panel, specify the following details:

   - **Namespace**
   - **ID**
   - **Name**
   - **Revision**

4. Select the **Data Type** from the list and then specify the required values for the selected data type, if available.

5. For certain data types, you can also specify the Key LOV by pasting the Key LOV ID in the **Key LOV** box.

6. Select the **Unit** applicable to the property from the list.



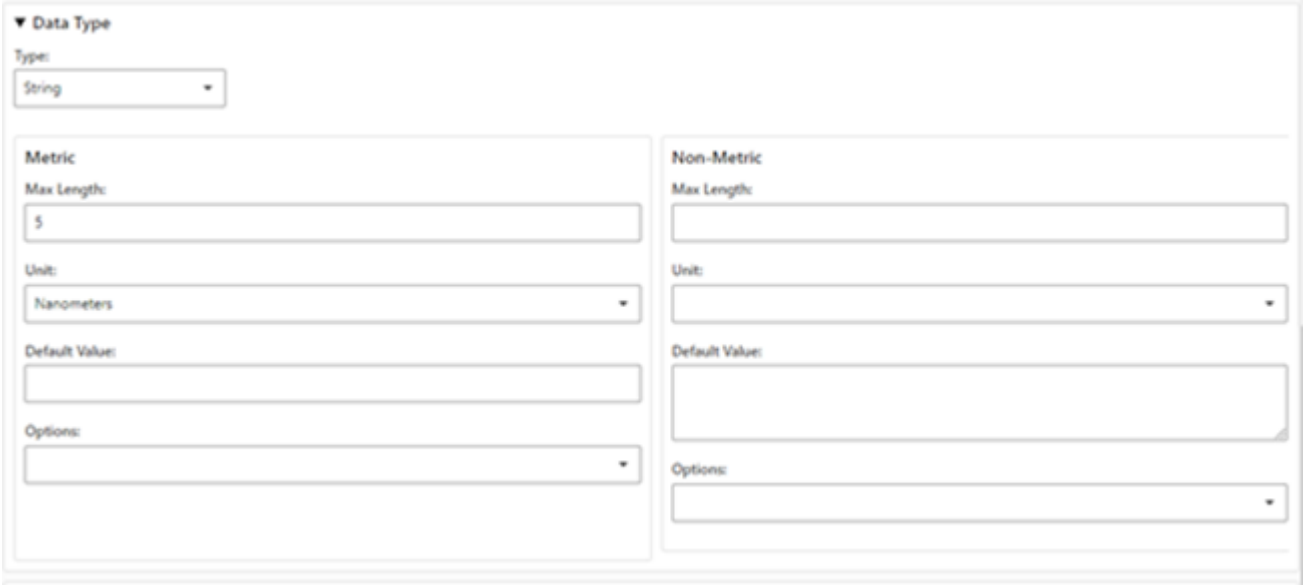
You can expand the arrow to select a specific unit type. For example, for **Length**, you can select from the different units of length such as, **Meters**, **Centimeters**, **Nanometers**, and so on.

7. Click **Add**.

8. To edit any property, click **Edit** .

   The editable cells are activated.

9. Enter or update the property values as required.



10. Save your changes by clicking **Save**. To cancel your edits, click **Cancel**.

Create a class

Create a class

Classes define the attributes that are stored for classified objects and determine which attributes are inherited by other classes. They are the primary building blocks of the Classification system. Workspace objects are classified by choosing a class from the hierarchy and providing values for the attributes of the class.

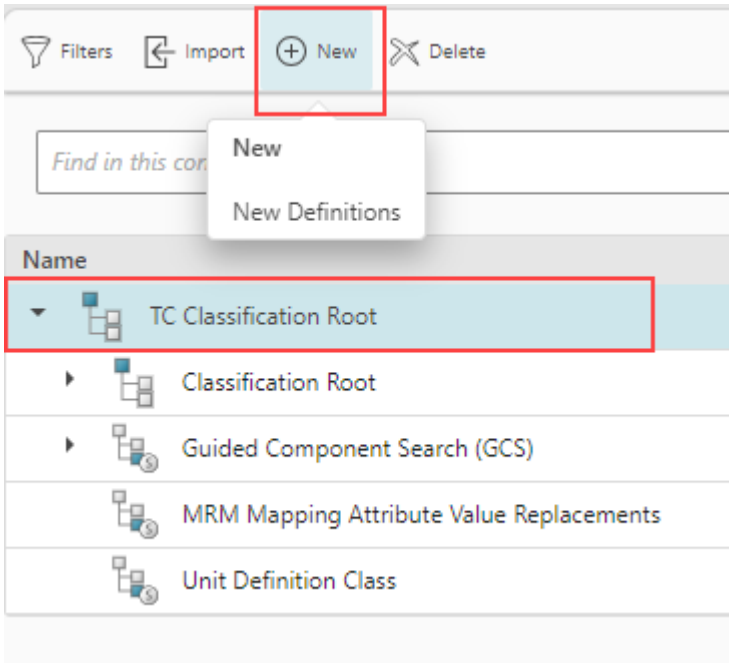
Procedure

1. On the **Classification Manager** dashboard, click the **Classes** tab.

2. Select a hierarchy under which you want to add a new class. You can do anyone of the following:

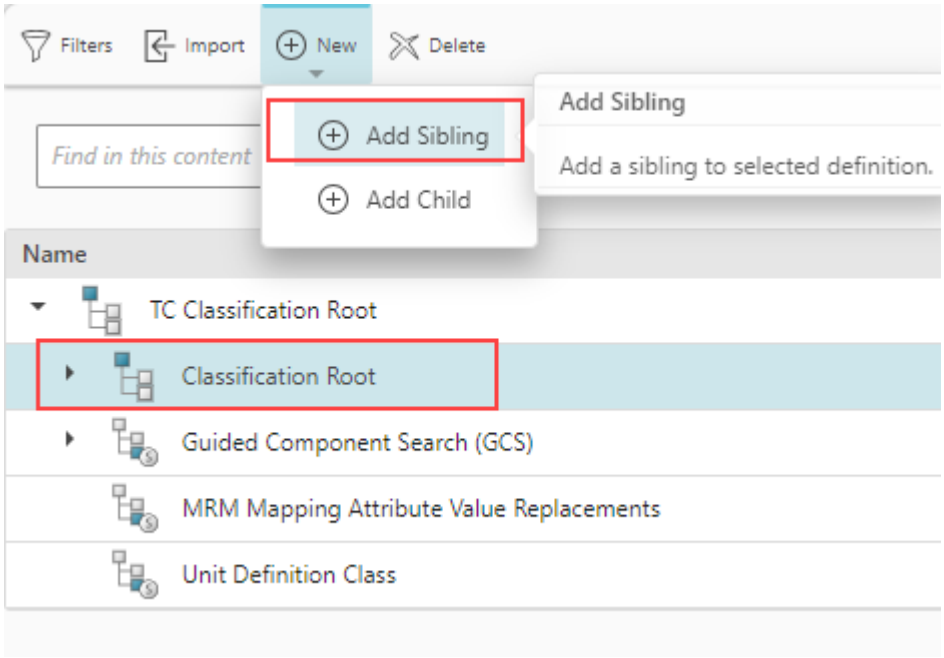
   - Add a class definition directly under the top level root class.

     To do this, select the top root class and click **New** ⊕.



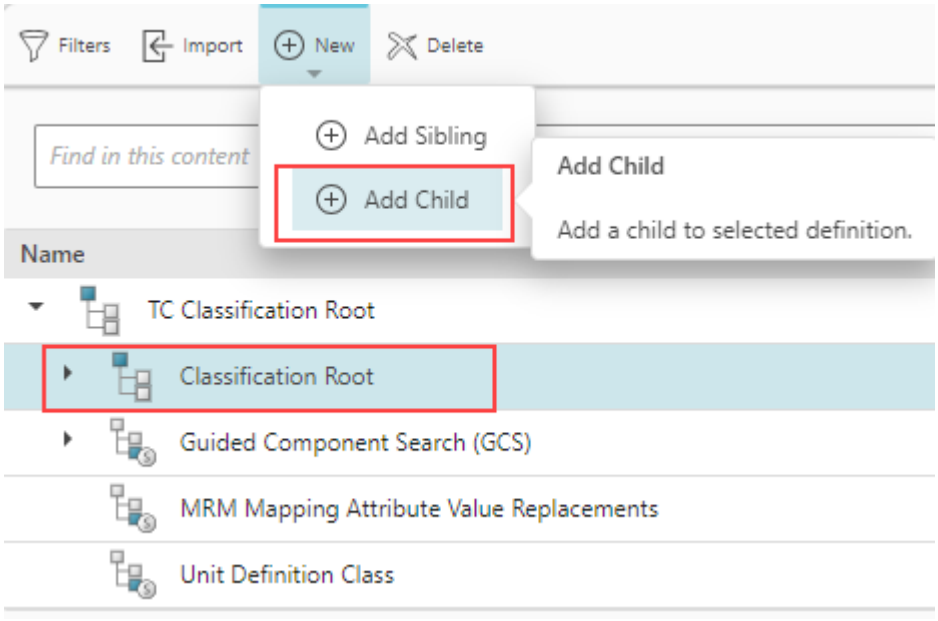
- Add a class definition as a sibling class at the same level as an existing class definition.

  To do this, select an existing class definition, click **New** ⊕ > **Add Sibling** ⊕.



- Add a class definition as a child class under an existing class definition.

  To do this, select an existing class definition, click **New** ⊕ > **Add Child** ⊕.



3. Specify the **Namespace**, **ID**, **Revision**, and **Name** for the class.

4. Select **Class Type** from the list.

5. Select **Metric**, **Non-Metric**, or **both** from the **Unit System** list to specify the system of measure to be applied to the attributes of the class.

6. Click **Add**.

   A new class is added within the selected hierarchy. You can click **Show All** in the **Overview** tab to view all the properties associated with the class.

   After creating a class, you must **add attributes to the definition of the class**. Attributes added to the class become inherited attributes of any child classes.

Create a class attribute

Attributes are either inherited or directly associated with the class. The **Inherited Attributes** list displays all the attributes that are inherited from any and all parent classes. Inherited attributes cannot be deleted or modified. You can, however, add attributes to the definition of the class. Attributes added to the class become inherited attributes of any child classes. A class can contain a maximum of 200 attributes, both inherited and local.

Procedure

1. Open the class definition to which you want to add the class attribute. Click **Class Attributes**.

2. Click **Add** ⊕.

3. In the **Add Class Attribute** panel, search for the attributes that you want to add to the class. Available search filter options are:

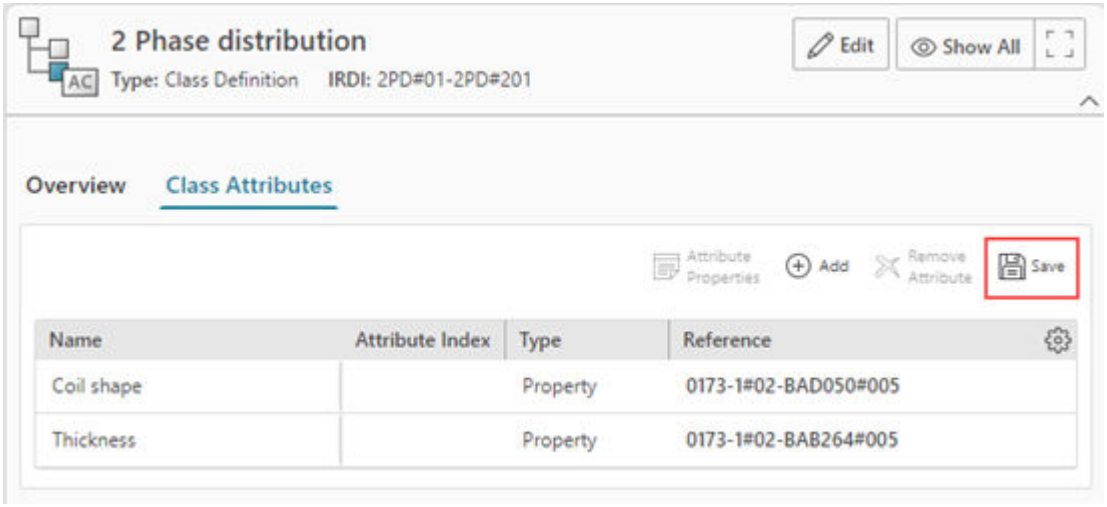
   - **Name**
   - **ID**
   - **Revision**
   - **Namespace**
   - **Status**

4. Specify the **Value** of the selected attribute.

5. Click **Search**.

6. Select the class attribute that you want to add from the search results and click **Add.** You can also select multiple class attributes to add at a time.

7. After all the attributes are added, click **Save**.



   The new class attributes are displayed automatically in the **Class Attributes** table.

8. To view additional attribute properties for each class attribute, select the class attribute and click **Attribute Properties**.

   If you select multiple class attributes and click **Attribute Properties**, you can compare attribute properties for all the selected class attributes.

9. To remove a class attribute, select the attribute and click **Remove Attribute**.

10. To edit any class attribute, click **Edit** .

    The editable cells are activated.

11. Enter or update the class attributes as required.

12. To save your changes, click **Save** . To cancel your edits, click **Cancel** .

Create a node

You can create a node with details such as node properties, application class, parent, and the details about the class and its properties. You can use the property groups navigator to get the property details within each property group.

However, if the nodes are not created automatically, you can do so for the basic data as can be done for the advanced classification as follows.

Procedure

1. On the **Classification Manager** dashboard, click the **Nodes** tab.

2. Click **New** > **Add Sibling** .

3. In the **Add Node** panel, select the **Node Type** from the list.

4. Specify the details in the remaining fields based on the selected node type.

5. Click **Add**.

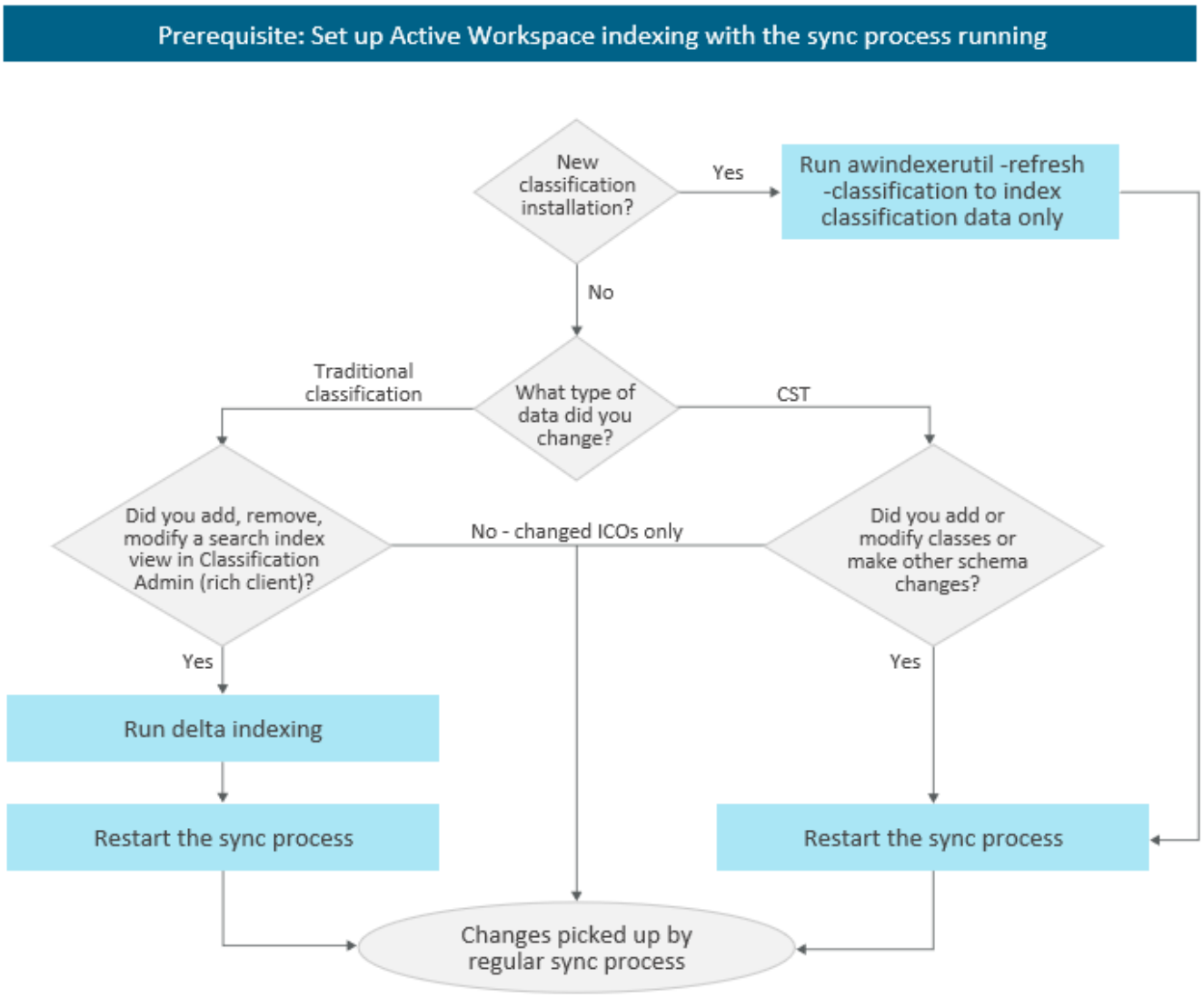
   A node is created, and the information about the node type and the application class is displayed under the **Overview** section.

   After you create or update the classification definitions for advanced classification in Classification Manager, you must release these definitions for the classification hierarchy to be available in the user interface for classifying the data.

Indexing classification data

About indexing classification data

Many changes to classification classes and data require indexing. There are several classification indexing options available to you. Classification indexing is based on the general Active Workspace indexing mechanism. As a prerequisite, classification assumes that general Active Workspace indexing is fully functional and synchronizing on a regular basis. The following workflow assists you with your indexing decisions:



Index classification data

If indexing is set up and running, use classification indexing to discover only the objects whose classification data has changed. This process marks these objects for indexing and they are picked up by the next scheduled round of regular indexing.

1. Ensure that regular indexing is running by entering the following in a command line:

   ```
   runTcFTSIndexer -task=objdata:test
   ```

2. Do one of the following:

   - Mark all classification data for indexing (or re-indexing):

     Run the following command:

```
awindexerutil -u=user -p=password -g=group -classification -refresh
```

Use this command if you have performed an initial installation of a classification feature or made substantial changes to your classification data and want to refresh all data. Running this command marks the following objects for indexing:

- Workspace objects classified in either traditional basic or advanced CST hierarchies
- Standalone classification objects (**cls0Object** and **cls0cstObject**)
- Library elements

If the release you are upgrading to includes new classification features, it generally also includes schema changes. During this upgrade procedure, all classification data is marked for re-indexing and included during your next indexing synchronization. This can cause the synchronization to take longer than usual.

- Index only changes made to the search index views in traditional basic classification:

  The search index views in traditional basic classification define which classes and properties are searchable. If you make any changes to these views, you can index only the changes made in the search index views. Changes in the search index view that require indexing are:

  - Adding or removing a search index view from a class
  - Adding or removing properties from a search index view

  a. Make properties searchable in the global search.

  b. (Optional) Run the **awindexerutil** utility:

  ```
  awindexerutil -u=user -p=password -g=group -delta
  -classification -dryrun
  ```

  Running this utility using the **dryrun** option displays the objects that will be marked for indexing in the next step. Scanning this output can assist you in troubleshooting the indexing of your data.

  c. Run the **awindexerutil** utility:

  ```
  awindexerutil -u=user -p=password -g=group -delta
  -classification
  ```

  This step marks changed data for indexing. Once marked, these objects are picked up in the next regular indexing synchronization flow.

Starting and stopping the indexing synchronization process

Before you begin classification indexing activities, you must stop the index synchronization process. The classification indexing mechanisms then mark objects for indexing. These objects are indexed the next time the synchronization process occurs.

After performing any classification indexing activities, is important to restart the synchronization process so the marked objects are picked up and indexed. Do this by scheduling regular syncing of the database with the **runTcFTSIndexer** with the **-task=objdata:sync** argument.

Administering classification

Classifying objects using clsutility

In addition to classifying objects in the user interface, you can perform this activity using **clsutility**. The utility requires a JSON file as input. The schema file used to create this request can be found in:

...*TC_DATA***\classification\json\schema\advanced***Classification_Save_PropertyRecords_Request_advanced.schema.json*

The following sample request called **classify.json** is based on this schema:

```
{
  "SchemaVersion": "1.1.0",
  "Locale": "en_US",
  "PropertyRecords": [
    {
      "ID": "027016/A",
      "ObjectType": "PR",
      "ClassDefinition": "SPLM-DL#01-000800#001",
      "UnitSystem": 1,
      "Properties": [
        {
          "ID": "SPLM-DL#02-000500#001",
          "Value": 3
        },
        {
          "ID": "SPLM-DL#02-000501#001",
          "Value": "Hi5"
        },
        {
          "ID": "SPLM-DL#02-000502#001",
          "Value": 4.57
        }
      ]
    }
```

```
    ]
  }
```

To classify the specified object (**027016/A**) in the **SPLM-DL#01-000800#001** class, enter the following command:

```
clsutility -u=user -p=password -g=group -create -classification_objects
-request="classify.json"
```

Classifying objects in multiple classes using clsutility

After an object is classified, you can classify it in other classes (*multiple classification*). This is done using the same clsutility command and schema file, but the request must include the **"IsMultipleClassification":true** parameter. For example:

```
{
  "SchemaVersion": "1.1.0",
  "Locale": "en_US",
  "PropertyRecords": [
    {
      "ID": "027016/A",
      "ObjectType": "PR",
      "ClassDefinition": "SPLM-DL#01-000800#001",
      "IsMultipleClassification": true,
      "UnitSystem": 1,
      "Properties": [
        {
          "ID": "SPLM-DL#02-000500#001",
          "Value": 3
        },
        {
          "ID": "SPLM-DL#02-000501#001",
          "Value": "Hi5"
        },
        {
          "ID": "SPLM-DL#02-000502#001",
          "Value": 4.57
        }
      ]
    }
  ]
}
```

Modify a classification

To modify one of the classifications, the request must include the UID of the property record (ICO), **PropertyRecordUID**, that you want to modify:

```
{
  "SchemaVersion": "1.1.0",
  "Locale": "en_US",
  "PropertyRecords": [
    {
      "ID": "027016/A",
      "ObjectType": "PR",
      "ClassDefinition": "SPLM-DL#01-000800#001",
      "PropertyRecordUID": "g_S588PW5c0mcA",
      "UnitSystem": 1,
      "Properties": [
        {
          "ID": "SPLM-DL#02-000500#001",
          "Value": 4
        },
        {
          "ID": "SPLM-DL#02-000501#001",
          "Value": "Hi5_new"
        },
        {
          "ID": "SPLM-DL#02-000502#001",
          "Value": 5.57
        }
      ]
    }
  ]
}
```

By default, objects cannot be classified in the same class multiple times. To enable this behavior, add the following to the JSON class definition:

```
"Options": [
        {
          "Name": "IsMultipleClassificationAllowed",
          "Value": true
        }
```

Deleting classifications

You must delete classifications in the user interface.

Finding the UID of an ICO

Use the **...***TC_DATA***classification****json****schema****advanced***Classification_FindRequest_advanced.schema.json* schema to create a JSON request. This request contains the ID of the classified object for which you want the UID of the classifying object, for example:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ObjectIDList": [
    {
      "ID": "027016/A"
    }
  ]
}
```

This request forms the input for the following command:

```
clsutility -u=user -p=password -g=group -find -classification_objects
-request=find_request.json -output=your-output-file.txt
```

The output of this command includes the value for **PropertyRecordUID**, which is the UID of the ICO that classifies the workspace object included in the request (in this case, **027016/A**).

Change the status of classification objects

You can only use classification objects that are in the released state.

Classification standard taxonomy supports three statuses:

- **Develop**
- **Released**
- **Retired**

Every class, property, and key-LOV is created with a status.

- When you create a new object and do not specify a release status, it is automatically assigned the **Develop** status. You can also specify the status explicitly:

This means the object is in a draft state. You can still modify it, for example, by adding or removing attributes in a class, but you cannot yet use it to classify data.

- Once the editing work is complete, you change the status of the object to **Released** using the **clsutility** command line utility or through the user interface.

- A **Released** class is available in the user interface to classify your data. You can **make minor changes in a released class without a new revision being created**.

- If you create a new revision of a class, the status of the previously existing revision is automatically changed to **Retired**. All classified objects in the **Retired** class are automatically moved to the **Released** class, with the exception of those that are already released.

- All revisions are imported with three digits, for example, **001**. If revisions are less than three digits, zeros are prefixed up to three digits so that **01** becomes **001**. If you then search for revision **01**, it is not found.

> Note:
>
> Nodes do not have a status and cannot be used to change the status of the application classes they reference.

After releasing a class, it is good practice to **create a base view definition of the class**. This improves the performance of the system and allows you to modify the order of presentation of the class properties.

Change the status of a class using the clsutility command

You can change the status of the class to **Released** using the **clsutility** command line utility.

Procedure

1. To change the status of the class run **clsutility** as follows:

   ```
   clsutility -u=user -p=password -g=group -update -status -request=JSON_file
   ```

   The JSON file must have a format similar to the following:

   ```
   {
       "SchemaVersion":"1.0.0",
       "Options":["recursive"],
       "Status":"Released",
       "ObjectIRDIList":
           [
           "TDOC#01-CLS001#001"
           ]
   }
   ```

   Running the utility using the **recursive** option changes the status of the class, as well as all the objects it references (properties, Key-LOV definitions, blocks, aspects) from **Develop** to **Released**.

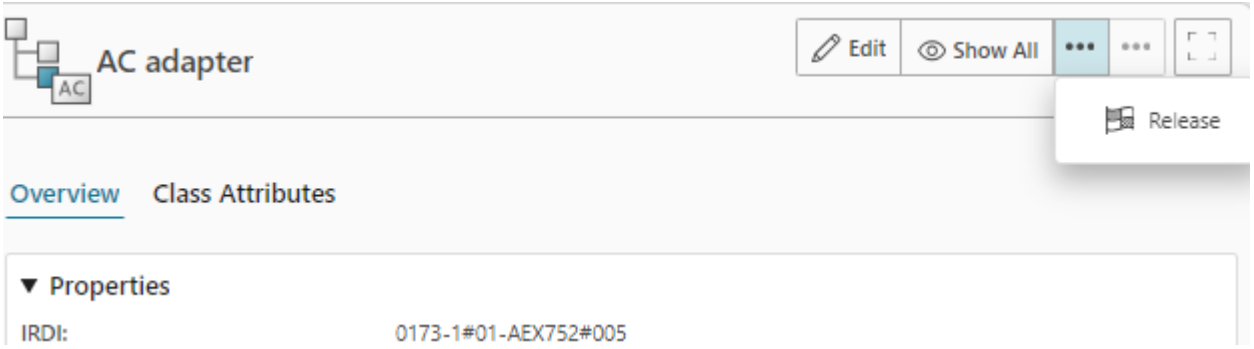
   > Note:
   >
   > Nodes do not have a status and cannot be used to change the status of the application classes they reference.

Change the status of a class from the user interface

You can change the status of the application class to **Released** from the user interface.

Procedure

1. To change the status of the class run **clsutility** choose **More Commands**••• > **Release**.



Results

The status of the class is changed to **Released** and the status of all the properties and Key-LOVs associated with the class are also updated to released.

Updating advanced classification objects without revisioning

In general, major changes in definition objects should result in a new revision of that object. However, many companies have workflows that involve making smaller tweaks and corrections to classes, properties, key-LOVs or nodes in their hierarchies. Revising the object each time a minor change is made may, in this case, result in a cluttered database with many meaningless revisions. To avoid this, you can make the following types of changes to an object definition without the object being revised:

- Updating or fixing spelling errors, for example, **Name** or **Description**
- Adding translations
- Adding key-LOV entries
- Deprecating key-LOV entries
- Adding class attributes
- Deprecating class attributes

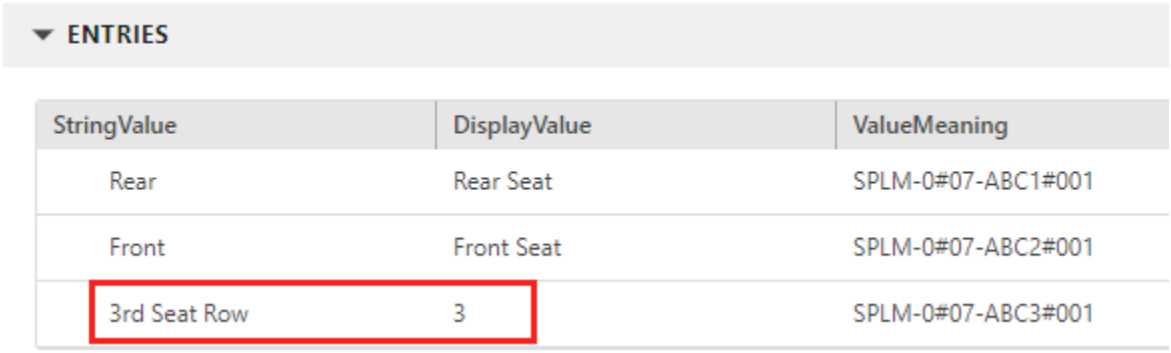
Note the following:

- Partial update is available beginning with schema 1.5.0 using the **SaveClassificationDefinitions_advanced.schema.json** schema file, found in **...***TC_DATA***\classification\json\schema\advanced**. Using this schema, you can both create definition objects and update others in the same JSON file.
- Only modifications that do not change the existing classification objects and their properties are allowed.

- Semantic changes are not permitted.

Example

You have a key-LOV, **SPLM-0#09-KEY002#001**, with the following entries:



You want to change the display name of the third key-LOV value from **3** to **Fold Away Seat**.

Using schema 1.5.0, you create the following JSON file:

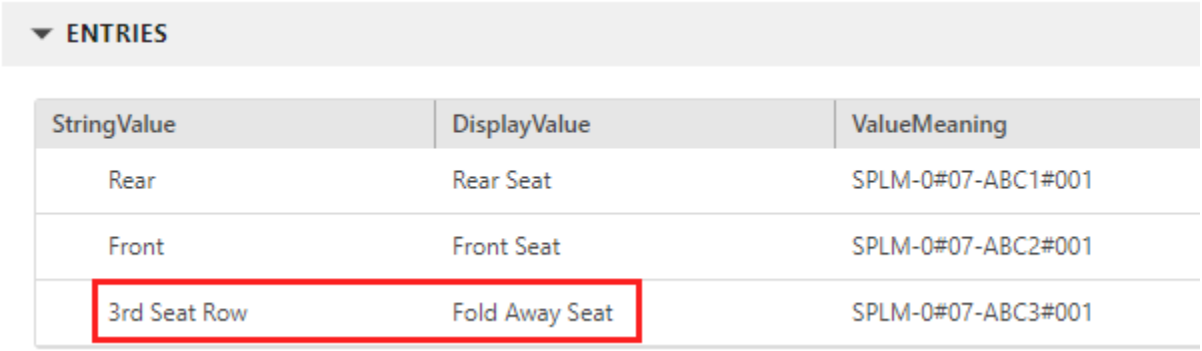
```
{
    "SchemaVersion": "1.5.0",
    "Locale": "en_US",
    "KeyLOVDefinitions": [{
        "Update": "SPLM-0#09-KEY002#001",
        "Values": {
            "LOVItems": [{
                    "Update": "3rd Seat Row",
                    "Values": {
                        "DisplayValue": "Fold Away Seat"
                    }
                }
            ]
        }
    }]
}
```

You can import this file using one of the following methods:

- In the user interface
- Using the **clsutility** with the following syntax:

  clsutility -u=*user* **-p=***password* **-g=***group* **-save -classification_definitions -request="***path-to-JSON-file***" -verbose -format**

When complete, the key-LOV is displayed as follows:



The revision of the key-LOV remains the same.

Deleting CST structures and data

Deleting objects is performed using **clsutility** with the **-delete** arguments. You may delete any of the following using the utility:

- Nodes
- Classification objects
- Classes
- Properties
- Key-LOVs

The utility requires a JSON file as input. The format for the JSON is found in the **schema directory**. The following example contains the request used to test (using the **dryrun** option) whether the **TDOC#01-CLS001#001** class and all the objects it references (including other block classes, key-LOVs, and properties) can be deleted:

```
{
   "SchemaVersion": "1.0.0",
   "Locale": "en_US",
   "Options": [ "recursive", "dryrun"],
   "ObjectIRDIList": [
       "TDOC#01-CLS001#001"
       ]
}
```

To actually delete the class and all the objects it references, remove the **dryrun** option in the JSON request file and run the **clsutility -delete** command again.

Note the following:

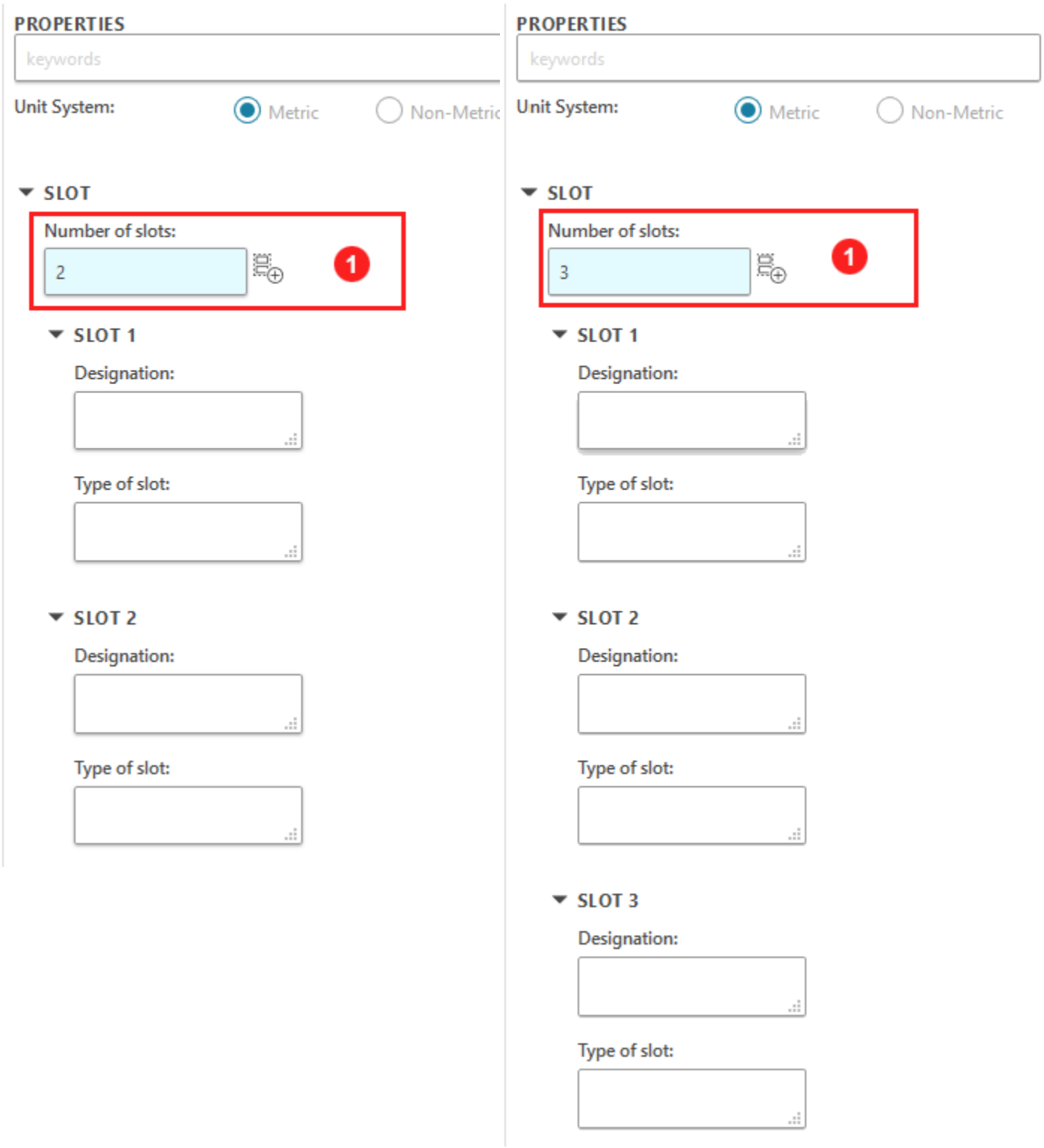
- To delete a branch in the classification hierarchy, delete the topmost class in the branch using the **recursive** argument. This deletes everything from that class downwards and prevents you from having to list all objects requiring deletion individually.

- You cannot delete a class in which objects are classified. You must first delete the classifications and then the classes.

About cardinality on property blocks

Cardinality on property blocks allows a user to specify the number of times that a property block class is used in an application class. For example, when designing a power supply for mobile devices, there is a class called **Power supply** containing two properties, **Designation** and **Type**. The number of times you can enter a designation and type depends on the value of a third property, **Number of slots**.

In the user interface, you first specify the number of slots 1, and then, for each slot, specify a designation and type.



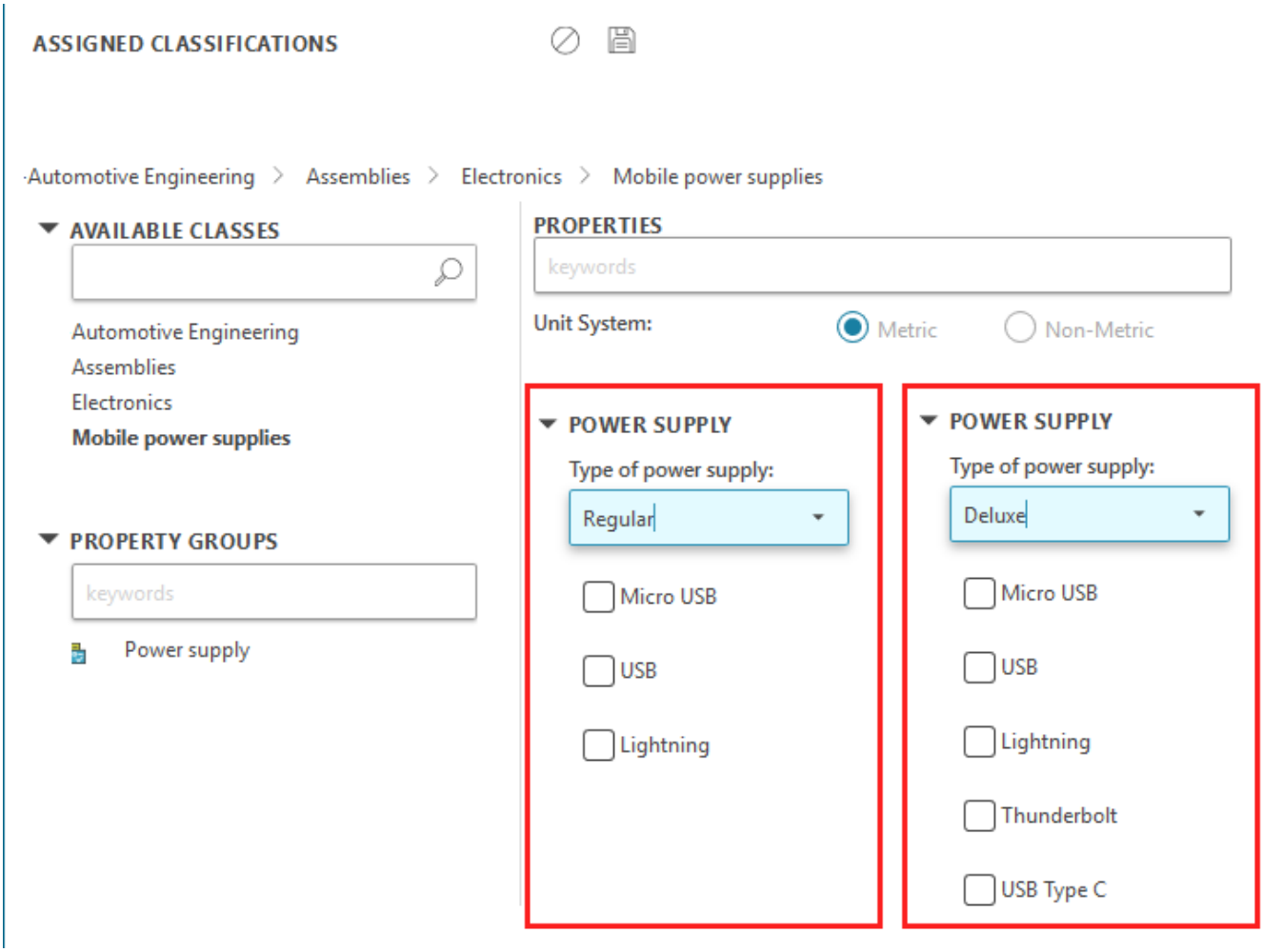
The **Number of slots** property is referred to as the *cardinality controller*.

About polymorphism

Polymorphism refers to the idea of displaying different sets of properties depending on the value of another property. For example, when designing a power supply for mobile devices, two types of power supplies are produced with the following connector types:

Regular	Deluxe
Micro USB	**Micro USB**
USB	**USB**
Lightning	**Lightning**
	USB-Type C
	Thunderbolt

The set (property block class) of connector types (properties) displayed depends on a property called **Type of power supply** (key-LOV). This property is referred to as the *polymorphism controller*. On the user interface, this is displayed as follows. A user can choose between a **Regular** or a **Deluxe** power supply and the properties displayed vary.

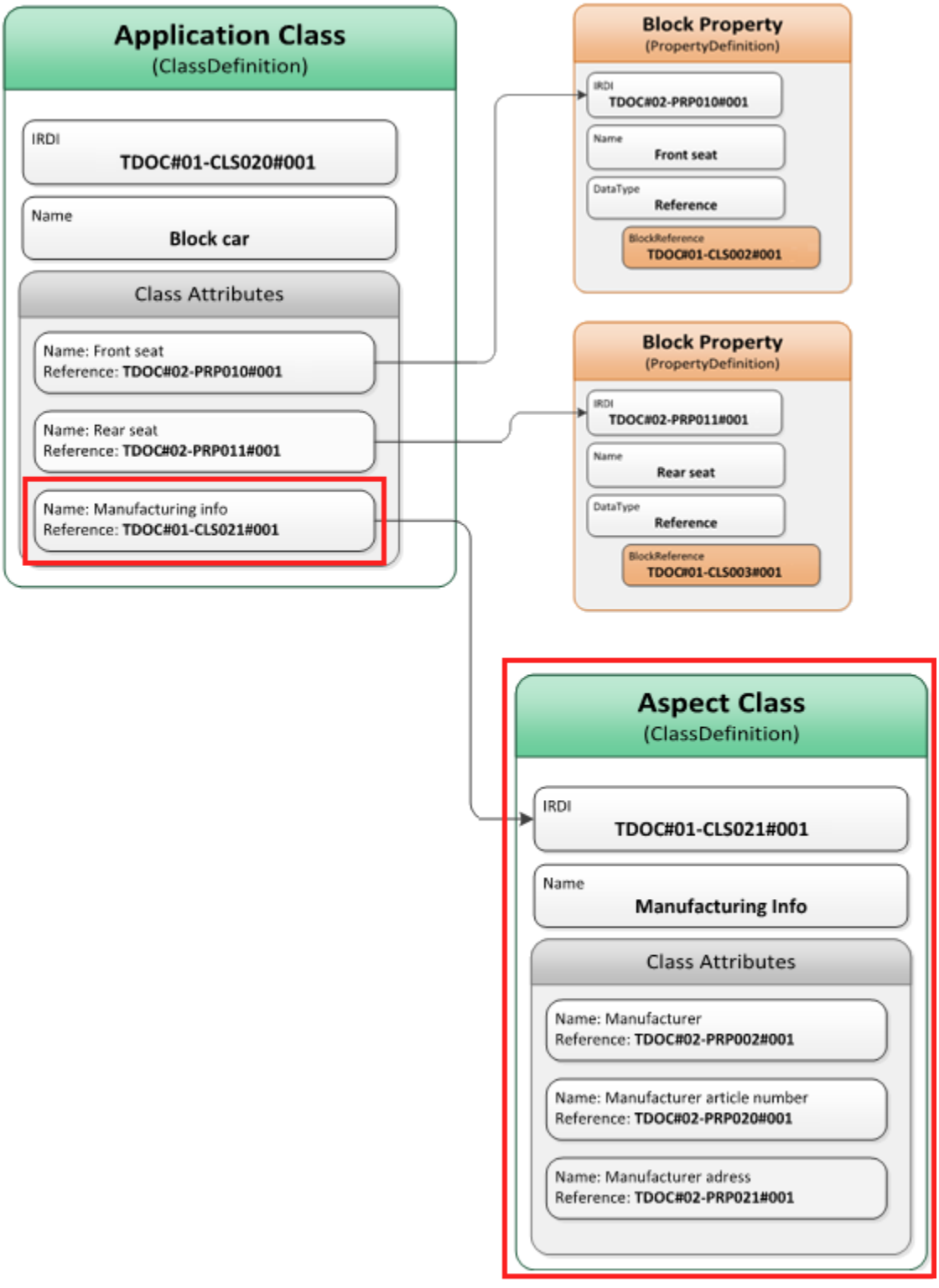


About aspects

An aspect is an **ECLASS object** that represents a group of properties. In classification standard taxonomy terms, an aspect is a kind of block class that can be referenced directly by another application class without going through a block property first.

Note that blocks, aspects, cardinality, and polymorphism are also supported in Advanced Classification without ECLASS taxonomy.

Example:



The two class attributes, **Front seat** and **Rear seat**, in the application class reference block properties that, in turn, reference property block classes. The third class attribute, **Manufacturer Info**, references the aspect class directly. The aspect class groups the properties pertaining to the manufacturer in one class.

The aspect class is represented in the JSON file as follows:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS021",
      "Revision": "001",
      "Name": "Front seat",
      "Status": "Develop",
      "IsAbstract": "true",
      "UnitSystem": 3,
      "ClassType": "Aspect",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP002#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP020#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP021#001"
        }
      ]
    }
  ]
}
```

The application class that references the aspect class is described as follows:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS020",
      "Revision": "001",
      "Name": "Car",
      "Status": "Develop",
      "IsAbstract": "False",
      "UnitSystem": 3,
      "ClassType": "Application Class",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP010#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP011#001"
        },
        {
          "Type": "Block",
          "Reference": "TDOC#01-CLS021#001"
        }
      ]
    }
  ]
}
```

Caution:

- When importing classes with aspects, you must first import the JSON file containing the aspect classes. Next, you import the application class that references the aspect. You cannot import these two types of classes using the same JSON file.
- You cannot reference an aspect class in a block class.
- Aspects cannot reference other aspects.

Using classification views to filter what information users can see

What are classification views?

Views are used to display all or a subset of the properties in a class in the user interface. There are two types of views used in classification classes:

Base view	A base view is used internally to cache data required to display the class, helping to improve performance. Additionally, you can use the base view to modify the order in which the properties of a class are displayed in the user interface. For best results, create a base view for each application class in a **Released** state. If no other view exists in the class, the user interface displays the properties contained in the base view.
User, group, or role view	These types of views are used to override the order of properties in the base view and limit the number of properties shown to different types of users based on their needs. For example, an Engineering group may need to see one group of properties, whereas a Logistics group needs to see a different group of properties. You can create views for specific groups, roles, or individual users.

> Note:
>
> You can create views only for application classes. Because of this, you can influence the order in which properties, blocks, and aspects are displayed in a class, but you cannot influence the order of properties within blocks or aspects.

Create base views in bulk

You can create base views in bulk using the **clsutility**.

Procedure

1. You can create base views for released applications as follows:

   - To create base views for every released application class, type:

     clsutility -create -base_view_definitions

     This command creates a base view definition for every application class in the **Released** state. If an application class already has a base view, it is skipped.

   - To create a base view for a specific application class, type:

     clsutility -create -base_view_definitions -class_definition=*IRDI of class*

     This command creates a base view for the application class with the specified IRDI.

   - To create a base view for all released application classes, regardless of whether they already have one:

     clsutility -create -base_view_definitions -force

This command creates a new base view definition for every application class in the **Released** state and overwrites any existing base view.

> Note:
>
> You cannot use the **-create -base_view_definitions** command to modify the order in which class properties are displayed in the user interface. To do this, you must use the **-create -view_definitions** command and specify the **ViewType** as **BaseView** in the JSON input file.

Create user, group, or role views

You create user, group, or role views by importing a JSON file that defines the properties that are displayed for each group, role, or user in the user interface. The properties of the class are displayed in their order of appearance in the JSON file (from top to bottom). You can include only properties in the views that already exist in the base view of the class. Hide attributes by excluding them in the JSON file for the view.

Procedure

1. To create a group, role, or user view containing properties in a specific order run the utility as follows:

   clsutility -create -view_definitions -request=*JSON file*

   The schema file describing the JSON format is found here:

   ...*TC_DATA***\classification**

   \json\schema\advanced*Classification_Save_ViewDefinitions_Request_advanced.schema.json*

   Refer also to the sample import file found here:

   ...*TC_DATA***\classification**
   \json\schema\advanced*Classification_Save_ViewDefinition_Request_sample.json*

   The format of the input file must resemble the following:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ViewDefinitions": [
    {
      "ClassDefinition": "TDOC#01-APPC04#001",
      "ViewType": "GroupView",
      "ViewId": "Engineering",
      "ViewAttributes": [
        {
```

```
            "IRDI": "TDOC#02-PDF015#001"
          },
          {
            "IRDI": "TDOC#02-PDF016#001"
          }
        ]
      },
      {
        "ClassDefinition": "TDOC#01-APPC04#001",
        "ViewType": "RoleView",
        "ViewId": "Designer",
        "ViewAttributes": [
          {
            "IRDI": "TDOC#02-PDF011#001"
          },
          {
            "IRDI": "TDOC#02-PDF014#001",
            "IsProtected": true,
            "IsRequired": true
          }
        ]
      },
      {
        "ClassDefinition": "TDOC#01-APPC04#001",
        "ViewType": "UserView",
        "ViewId": "JSmith",
        "ViewAttributes": [
          {
            "IRDI": "TDOC#02-PDF012#001"
          },
          {
            "IRDI": "TDOC#02-PDF013#001"
          }
        ]
      }
    ]
  }
```

Notice that you can specify whether a property value must be entered for a property (**IsRequired**) or whether it is protected and cannot be modified in the user interface (**IsProtected**).

To create a base view to reorder its properties, the input must resemble the following:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ViewDefinitions": [
    {
      "ClassDefinition": "TDOC#01-APPC04#001",
```

```
        "ViewType": "BaseView",
        "ViewId": "BaseView",
        "ViewAttributes": [
          {
            "IRDI": "TDOC#02-PDF013#001"
          },
          {
            "IRDI": "TDOC#02-PDF015#001"
          },
          {
            "IRDI": "TDOC#02-PDF011#001",
            "IsProtected": true
          },
          {
            "IRDI": "TDOC#02-PDF014#001",
            "IsRequired": true
          },
          {
            "IRDI": "TDOC#02-PDF012#001"
          },
          {
            "IRDI": "TDOC#02-PDF016#001"
          }
        ]
      }
    ]
}
```

Extract existing view definition

You can extract an existing view definition from the database using the clsutility command.

Procedure

1. Run the following command:

   clsutility -find -view_definitions -request=*JSON-schema-version*

   The JSON schema file for the **-find** command is found in:

   ...*TC_DATA***\classification**
   \json\schema\advanced*Classification_Find_View_Request_advanced.schema.json*

   Refer also to the sample file found here:

   ...*TC_DATA***\classification**
   \json\samples\advanced*Classification_Find_ViewDefinition_Request_sample.json*

The JSON file must resemble the following:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ViewDefinitions": [
    {
      "ClassDefinition": "TDOC#01-APPC10#001"
    },
    {
      "ClassDefinition": "TDOC#01-APPC12#001",
      "ViewType": "GroupView"
    },
    {
      "ClassDefinition": "TDOC#01-APPC09#001",
      "ViewId": "JSmith"
    },
    {
      "ClassDefinition": "TDOC#01-APPC08#001",
      "ViewType": "GroupView",
      "ViewId": "Engineering"
    },
    {
      "ViewType": "UserView",
      "ViewId": "JSmith"
    },
    {
      "ViewId": "MNancy"
    },
    {
      "ClassDefinition": "TDOC#01-APPC11#001",
      "ViewType": "BaseView",
      "ViewId": "BaseView"
    }
  ]
}
```

It is not necessary to uniquely identify each view that you want to display. It is sufficient to just request; for example, all the views for a particular class:

```
{
      "ClassDefinition": "TDOC#01-APPC10#001"
    }
```

Alternatively, you could request all user views for a particular user:

```
 {
      "ViewType": "UserView",
```

```
  "ViewId": "JSmith"
}
```

List the class descriptor for a view

Whereas a class definition describes only the class, or a key-LOV definition only one key-LOV, the class descriptor compiles all the definitions required to describe an ICO. Each node of the hierarchy shares a class descriptor. The class descriptor provides a view on the data from the classification consumer perspective.

Procedure

1. To obtain the class descriptor for a particular view run the following command:

   clsutility -get -class_descriptor -request=*JSON file name*

   The schema file explaining the required JSON syntax is found in:

   ...*TC_DATA***\classification**
   \json\schema\advanced*Classification_Get_ClassDescriptor_Request_advanced.schema.json*

Delete view definitions

You can delete a view definition using the clsutility.

Procedure

1. To delete a view definition, type:

   clsutility -find -view_definitions -request=*JSON file name*

   When deleting, the JSON request file uses the same schema as the JSON find request.

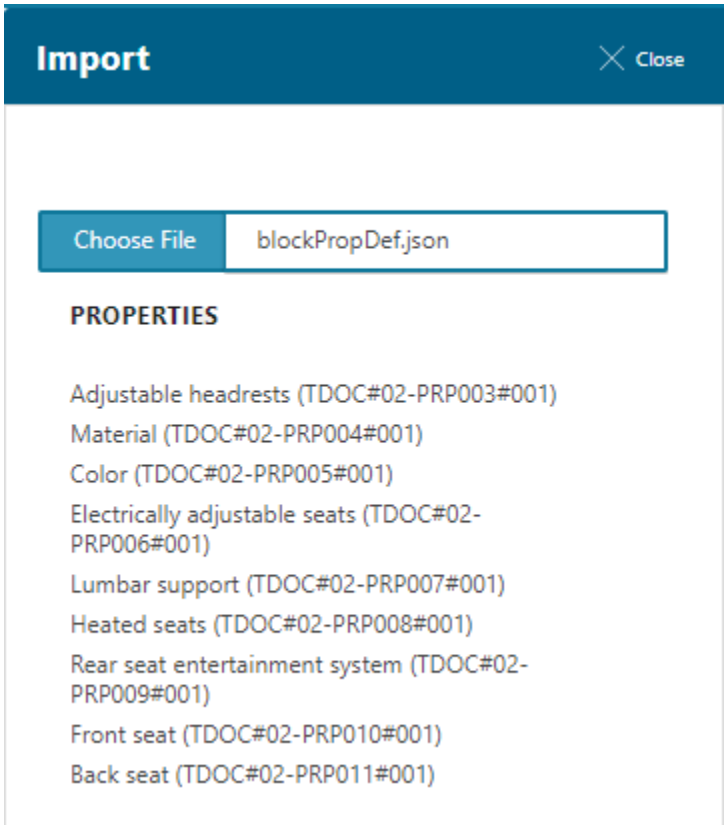
   > Note:
   >
   > Be careful when deleting base views. For performance reasons, a released class should always contain a base view. It is advisable to overwrite an undesired base view with a corrected version instead of deleting it.

   Deleting a class definition deletes all views belonging to the class.

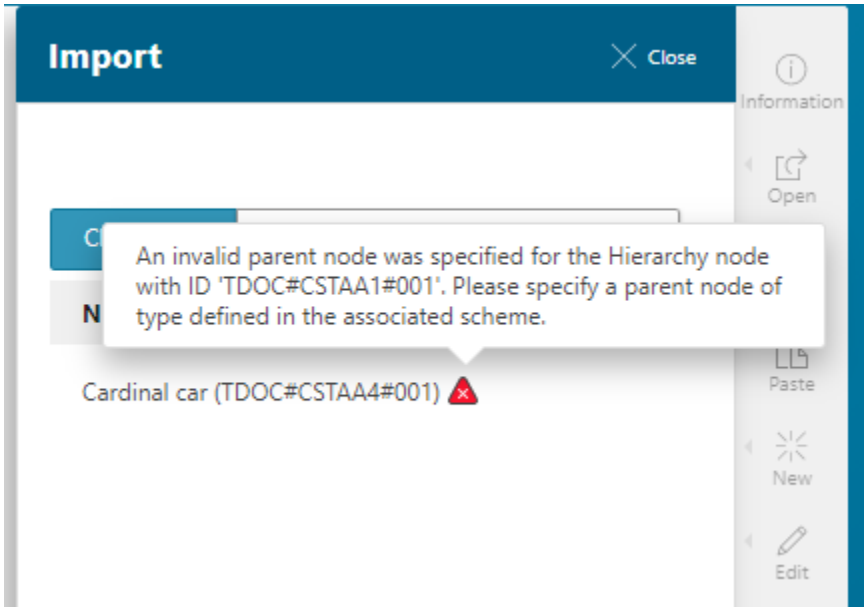
Importing and exporting classification data

Importing hierarchy definitions with the Classification Manager

For definition files based on schemas prior to version 1.5, you can import the definitions directly in the Teamcenter user interface. Use the **Import** button on the **Dashboard** tab or the **Import** command on any of the other tabs in **Classification Manager** to import key-LOV, property, class, and node definitions. They must always be **imported in the correct order**. After you upload the JSON file, you receive an overview of the objects you are about to import.



If there is an error in the files that you import, the error message provides assistance.



Tip:

It does not matter on which tab you choose to import, you can import any type of hierarchy definition on any of the tabs.

Import classified data for advanced classification

You can import classified data in BMEcat XML or JSON format in the user interface. The XML files and supporting documents (for example, images) must be available in a local directory that you compress to a ZIP file. The import searches the compressed for the correct input file.

1. Compress the directory structure of the data in a ZIP file.

   If you download BMEcat files from the Siemens Industry Mall, these are already in the appropriate zipped format.

2. In your **Home** folder, choose **More Commands** ••• > **Import/Export** > **Import Classification Data**.

3. In the **Choose File** box, select the ZIP file containing the data.

   Teamcenter searches inside the compressed file for all eligible import files and presents them to you in a list.

4. In the **Select file to import** box, select the XML file containing the data.

   If you import BMEcat data, there may be several versions of the XML file in different languages. These are not localized values. If you import, for example, the English version, and then

subsequently import the German version, the German version overwrites the English values. Each of these language files represent master locales.

When the import is complete, you are notified in the **Alert** area where you can view the details of the import. As soon as the data is indexed, it is available for searching.

This feature requires that both the Subscription Manager and Dispatcher Client are running. If the Subscription Manager is not running, no notification is displayed in the **Alert** area, but the operation still occurs in the background.

If the imported data includes classification objects only, at import they are automatically attached to existing objects in the database if they have the same ID. This automatically classifies the existing objects or updates an existing classification on previously classified objects. If there is no existing object with the same ID in the database, a new object is created.

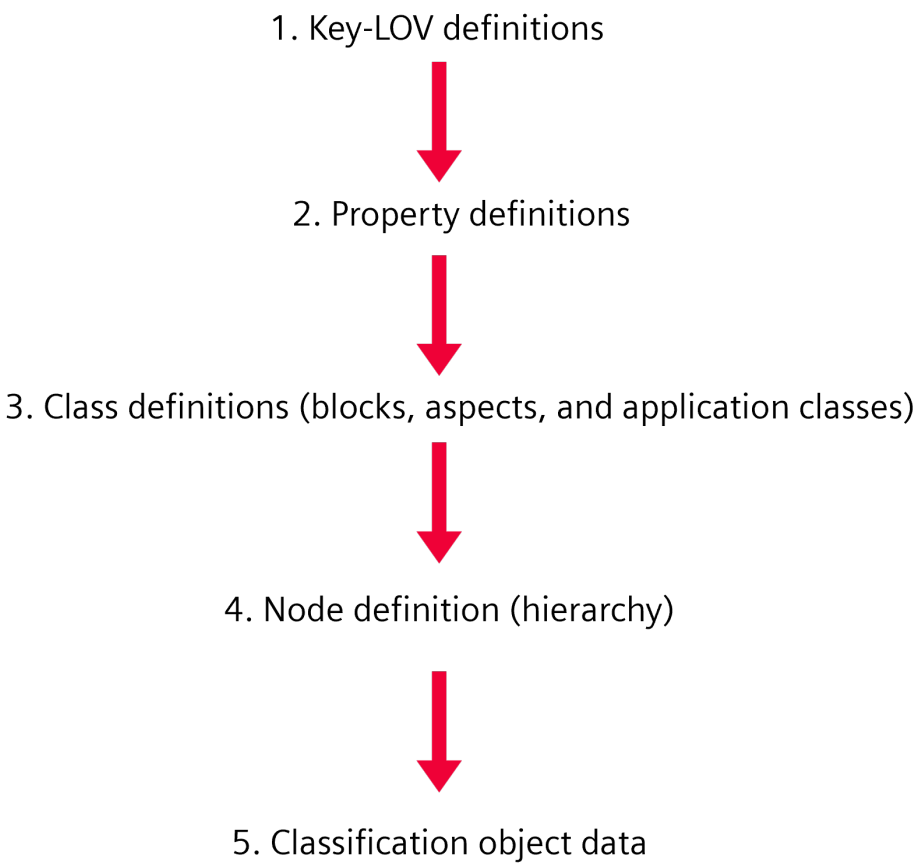
Importing JSON files using the clsutility command

You create a CST hierarchy by importing JSON files containing both hierarchy and data. The schema files defining the JSON format and example files are in the following directory:

...*TC_DATA*\classification\json\schema\advanced

The schema used to import classification definitions is evolving over time. Prior to version Teamcenter 14.1 (schema 1.5.0), key-LOVs, properties, class, and nodes had to be defined in separate JSON files and imported in the order listed. Using schema 1.5.0, you can import all types of class definitions in a single JSON file. This schema also supports importing older object definition JSON files with no need to rewrite them. You can define new objects or update existing ones in the same JSON file.

When importing, you can only reference objects that are already in the database. When importing classes that reference other classes (for example, when importing aspects), you must import the referenced classes first and then import the referencing classes. The definition file is read from top to bottom. If you import individual files, the order of import is as follows:



Data is imported from JSON files using the following methods:

- **Use the Classification Manager to import JSON files** in the user interface.
- Use **clsutility** to import a definition file based on any schema up to and including version 1.5.

  clsutility -u=*user* **-p=***password* **-g=***group* **-save -classification_definitions -request="***path-to-JSON-file***"**
- Use **clsutility** to import object individual definition files.

  The syntax of this utility is:

  clsutility-u=*user* **-p=***password* **-g=***group -operation input/output file path*

  If using schema 1.5, import the definition file using the following syntax:

  clsutility -u=*user* **-p=***password* **-g=***group* **-create -keylov_definitions -request="***path-to-JSON-file***"**

For more information about the available commands and their exact syntax, type **clsutility -h** for the top-level help and **clsutility -***command-name* **-h** for information about each available command.

The following table lists the supported objects and an example of the **clsutility** statements used to import these object types.

Object	Operation	clsutility command
Key-LOV	Import	**clsutility -u=***user* **-p=***password* **-g=***group* **-create -keylov_definitions -request="***path-to-JSON-file***"**
Property definition	Import	**clsutility -u=***user* **-p=***password* **-g=***group* **-create -property_definitions -request="***path-to-JSON-file***"**
Class definition	Import	**clsutility -u=***user* **-p=***password* **-g=***group* **-create -class_definitions -request="***path-to-JSON-file***"**
Nodes	Import	**clsutility -u=***user* **-p=***password* **-g=***group* **-create -node_definitions -request="***path-to-JSON-file***"**
Application data (ICOs)	Import	**clsutility -u=***user* **-p=***password* **-g=***group* **-create -classification_objects -request="***path-to-JSON-file***"**

Note:

There are pairs of very similar commands in the **clsutility**: **classification_objects** and **classification_object**, or **node_definitions** and **node_definition**. Be sure to use the plural forms of these commands (**objects** and **definitions**) for all CST imports. The singular refers to traditional classification objects.

Preparing JSON files for import (example)

Importing the hierarchy and classes of car seats

This example walks you through creating and importing the JSON files required to describe a class containing attributes that allow you to configure the seats in a car. The example consists of several levels of complexity:

- Classify a car in a class containing simple properties, for example, a key-LOV with the type of seat (front seat or back seat).
- Classify a car in a class containing reusable sets of properties (blocks).
- Specify the number of seats when classifying the car (cardinality).
- Specify the attributes shown depending on the type of seat selected (polymorphism).
- Specify the attributes shown for each seat when specifying multiple seats (cardinality and polymorphism).

- Import a property record into one of the newly created classes.

For each of these examples, the JSON files are explained. Once created, the JSON files can be imported using the Classification Manager or, alternatively, with **clsutility**. The **clsutility** commands required to import the JSON files are also listed **here**.

To verify that the definitions in the following examples are correct or to create additional definitions, consult the following schema files found in **...***TC_DATA***\classification\json\schema\advanced**:

- **Classification_Save_KeyLOVDefinitions_Request_advanced.schema.json**
- **Classification_Save_PropertyDefinitions_Request_advanced.schema.json**
- **Classification_Save_ClassDefinitions_Request_advanced.schema.json**
- **Classification_Save_NodeDefinitions_Request_advanced.schema.json**
- **Classification_Save_PropertyRecords_Request_advanced.schema.json**

There are many online JSON validators to assist you in creating JSON files with the proper syntax. We recommend using any of these to verify the JSON files that you create prior to importing them.

Note the following points about these examples:

- The definitions are imported with a release status of **Develop**. This ensures that you can still modify them if required. When you are satisfied with the structure and classes, you must **change the status to Released**. Objects can be classified only in **Released** classes.
- The **TDOC** name space is used to differentiate the structures imported in this example from other structures that you use. Creating your own name space helps when creating queries. When searching for your newly-imported objects, you can filter the tabs in Classification Manager to only show those with **Namespace** = **TDOC**.

Import a class with simple properties

The simplest use case is to import a class containing properties. This example shows how to import a **Car** class containing three properties:

- **Type of seat**

  A key-LOV with two entries: **Front** and **Back**

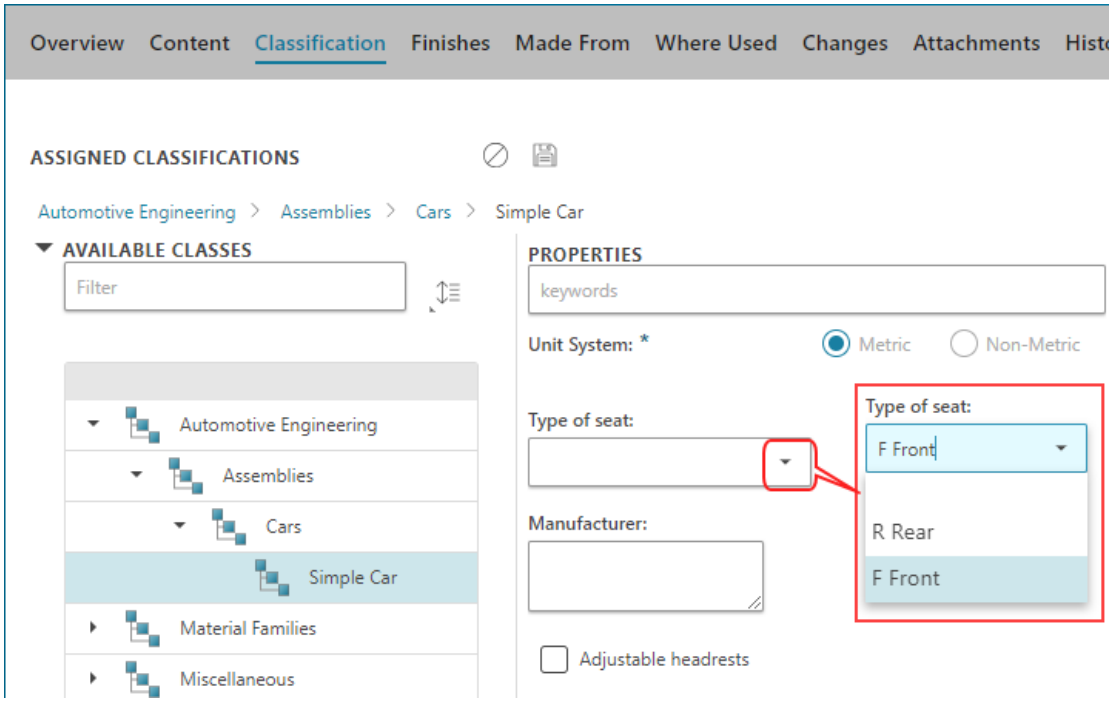
- **Manufacturer**

  A simple string property

- **Adjustable headrests**

  A Boolean property

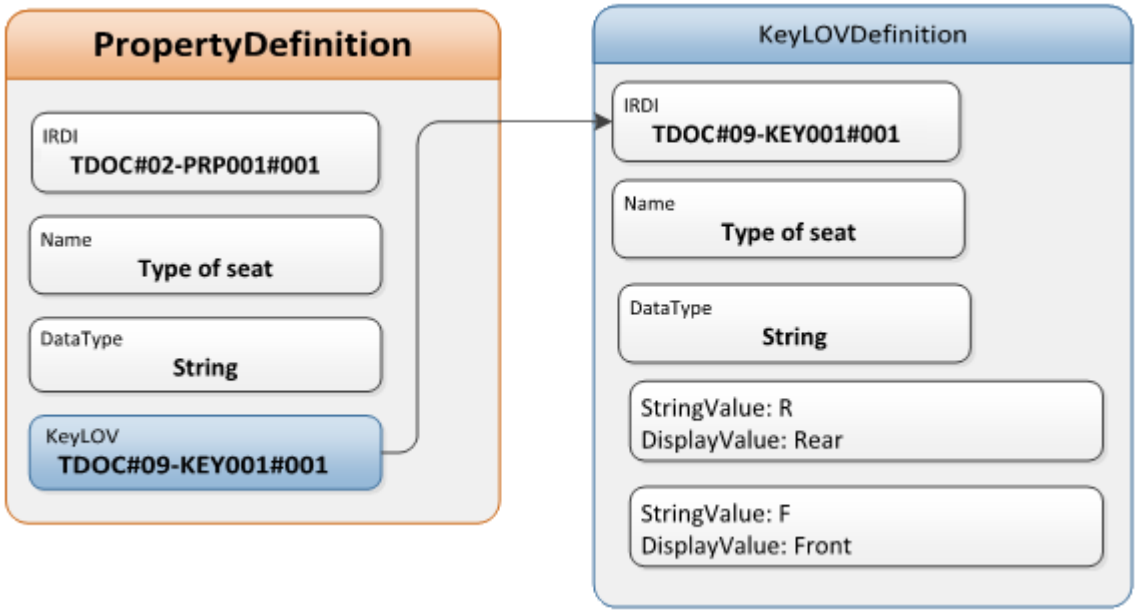
In the user interface, this class is displayed as follows:



The order when importing this class is to first import the key-LOV definitions, then the property definitions, then the class definition, and finally the node definition so that the class is visible in the classification hierarchy.

1. Import the key-LOV definition.

   In classification standard taxonomy, a key-LOV definition is separate from its property definition.



In this example, there is one key-LOV whose JSON definition is stored in a file titled **simpleKeylovDef.json**:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "KeyLOVDefinitions": [
    {
      "ObjectType": "09",
      "Namespace": "TDOC",
      "ID": "KEY001",
      "Revision": "001",
      "Name": "Type of seat",
      "Status": "Develop",
      "LOVItems": {
        "DataType": "String",
        "LOVStringItems": [
          {
            "StringValue": "R",
            "DisplayValue": "Rear Seat"
          },
          {
            "StringValue": "F",
            "DisplayValue": "Front Seat"
          }
        ]
      }
    }
  ]
}
```

2. Import the property definitions.

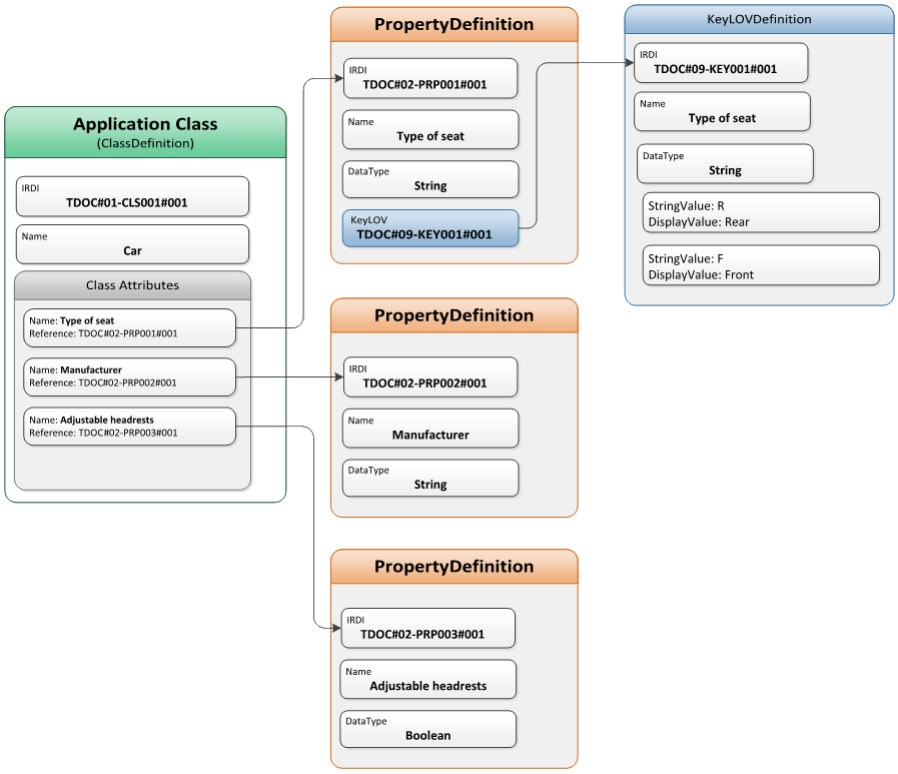
In this example, there are three properties to import, one of which (**Type of seat**) is a key-LOV that references the key-LOV definition imported in the previous step. The JSON definition for the properties is stored in a file titled **simplePropDef.json**:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "PropertyDefinitions": [
        {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP001",
      "Revision": "001",
      "Status": "Develop",
      "Name": "Type of seat",
      "DataType": {
        "Type": "String",
        "KeyLOV": "TDOC#09-KEY001#001"
      }
    },
     {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP002",
      "Revision": "001",
      "Name": "Manufacturer",
      "Status": "Develop",
      "DataType": {
        "Type": "String"
      }
    },
        {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP003",
      "Revision": "001",
      "Name": "Adjustable headrests",
      "Status": "Develop",
      "DataType": {
        "Type": "Boolean"
      }
    }


  ]
}
```

3. Import the class definition.

The **Car** class contains only three properties: **Type of seat**, **Manufacturer**, and **Adjustable headrests**:



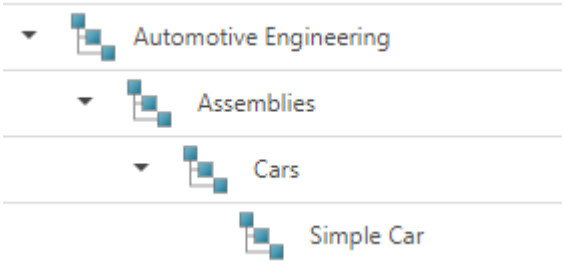
The JSON file, named **simpleApplicationClassDef.json**, consists of the following:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS001",
      "Revision": "001",
      "Name": "Car",
      "Status": "Develop",
      "IsAbstract": false,
      "UnitSystem": 3,
      "ClassType": "Application Class",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP001#001"
        },
        {
```

```
            "Type": "Property",
            "Reference": "TDOC#02-PRP002#001"
          },
          {
            "Type": "Property",
            "Reference": "TDOC#02-PRP003#001"
          }
        ]
      }
    ]
  }
```

4. Import the node definition that makes the class visible.

   A nested hierarchy is required to organize the classes.



   To import this hierarchy, create the following JSON file and name it **simpleNodeDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "NodeDefinitions": [
    {
      "Namespace": "TDOC",
      "ID": "CSTAE0",
      "Revision": "001",
      "Name": "Automotive Engineering"
    },
    {
      "Namespace": "TDOC",
      "ID": "CSTAA0",
      "Revision": "001",
      "Parent": {
        "Namespace": "TDOC",
        "ID": "CSTAE0",
        "Revision": "001"
      },
      "Name": "Assemblies"
    },
```

```
    {
      "Namespace": "TDOC",
      "ID": "CSTAA1",
      "Revision": "001",
      "Parent": {
        "Namespace": "TDOC",
        "ID": "CSTAA0",
        "Revision": "001"
      },
      "Name": "Cars"
    },
    {
      "Namespace": "TDOC",
      "ID": "CSTAA2",
      "Revision": "001",
      "Parent": {
        "Namespace": "TDOC",
        "ID": "CSTAA1",
        "Revision": "001"
      },
      "Name": "Simple Car",
      "ApplicationClass": {
        "Namespace": "TDOC",
        "ID": "CLS001",
        "Revision": "001"
      }
    }
  ]
}
```

5. Import the definitions into the database using **Classification Manager** or with **clsutility commands**.

   You must import the JSON files in the following order:

   a. **simpleKeylovDef.json**

   b. **simplePropDef.json**

   c. **simpleApplicationClassDef.json**

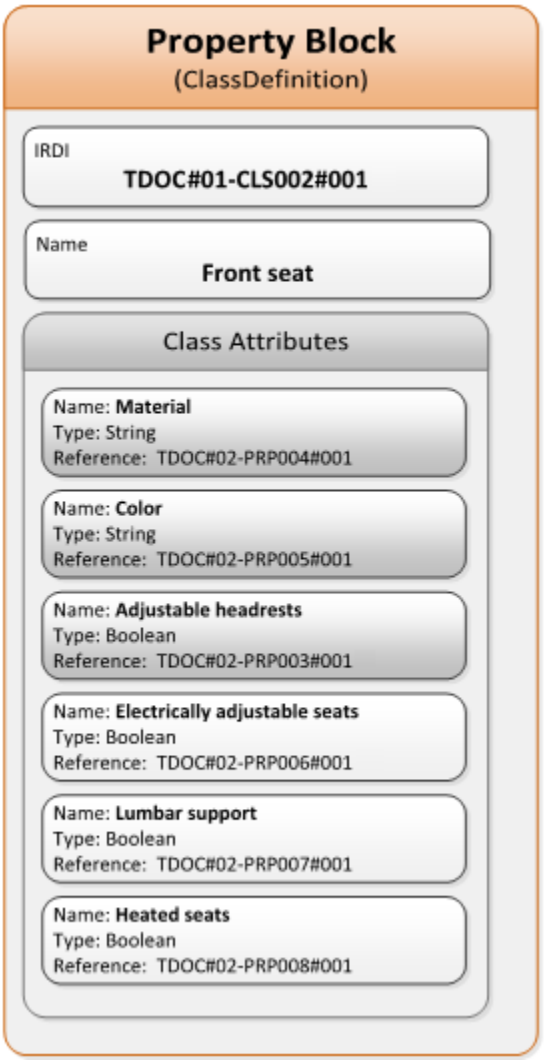
   d. **simpleNodeDef.json**

Import a class with a reusable set of properties (blocks)

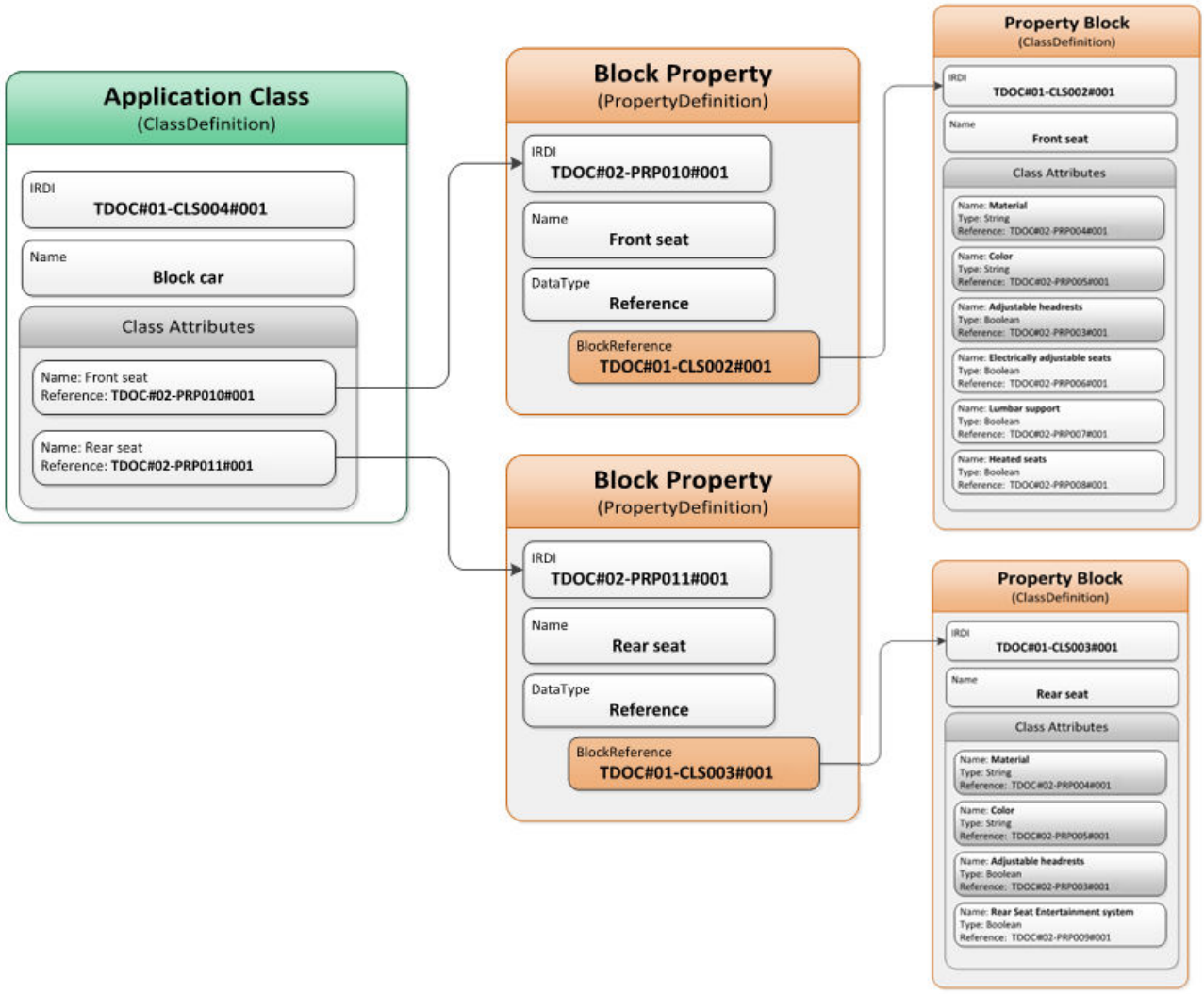
Frequently, there is a group of properties that is always used together. For example, the following set of properties describe a front seat and a rear seat:

Front seat	Rear seat
Material	Material
Color	Color
Adjustable headrests	Adjustable headrests
Electrically adjustable seats	Rear seat entertainment
Lumbar support	
Heated seats	

Each time you create a class, you could manually add each of these properties to the class to describe each of these seat types. However, it is easier to group them and select a single property called **Front seat** that contains all the required properties. This set of properties is called a property block. A property block is a special kind of class.



Property blocks are referenced by block properties which, in turn, are referenced in the application class containing the block properties:



To create the JSON files required to import these classes and properties:

1. Create the individual property definitions.

   Create the following JSON file and save it in a file named **blockPropDef.json** as follows:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "PropertyDefinitions": [
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP004",
```

```
    "Revision": "001",
    "Status": "Develop",
    "Name": "Material",
    "DataType": {
      "Type": "String"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP005",
    "Revision": "001",
    "Name": "Color",
    "Status": "Develop",
    "DataType": {
      "Type": "String"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP003",
    "Revision": "001",
    "Name": "Adjustable headrests",
    "Status": "Develop",
    "DataType": {
      "Type": "Boolean"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP006",
    "Revision": "001",
    "Name": "Electrically adjustable seats",
    "Status": "Develop",
    "DataType": {
      "Type": "Boolean"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP007",
    "Revision": "001",
    "Name": "Lumbar support",
    "Status": "Develop",
    "DataType": {
      "Type": "Boolean"
```

```
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP008",
    "Revision": "001",
    "Name": "Heated seats",
    "Status": "Develop",
    "DataType": {
      "Type": "Boolean"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP009",
    "Revision": "001",
    "Name": "Rear seat entertainment system",
    "Status": "Develop",
    "DataType": {
      "Type": "Boolean"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP010",
    "Revision": "001",
    "Name": "Front seat",
    "Status": "Develop",
    "DataType": {
      "Type": "Reference",
      "BlockReference": "TDOC#01-CLS002#001"
    }
  },
  {
    "ObjectType": "02",
    "Namespace": "TDOC",
    "ID": "PRP011",
    "Revision": "001",
    "Name": "Back seat",
    "Status": "Develop",
    "DataType": {
      "Type": "Reference",
      "BlockReference": "TDOC#01-CLS003#001"
    }
  }
```

```
    ]
  }
```

The last two properties, **PRP010** and **PRP011** reference the two property block classes that hold the sets of properties.

2. Create the class definitions, including the property block classes.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS002",
      "Revision": "001",
      "Name": "Front seat",
      "Status": "Develop",
      "IsAbstract": true,
      "UnitSystem": 3,
      "ClassType": "Block",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP004#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP005#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP003#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP006#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP007#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP008#001"
        }
      ]
    },
```

```
{
  "ObjectType": "01",
  "Namespace": "TDOC",
  "ID": "CLS003",
  "Revision": "001",
  "Name": "Back seat",
  "Status": "Develop",
  "IsAbstract": true,
  "UnitSystem": 3,
  "ClassType": "Block",
  "ClassAttributes": [
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP004#001"
    },
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP005#001"
    },
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP003#001"
    },
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP009#001"
    }
  ]
},
{
  "ObjectType": "01",
  "Namespace": "TDOC",
  "ID": "CLS004",
  "Revision": "001",
  "Name": "Car",
  "Status": "Develop",
  "IsAbstract": false,
  "UnitSystem": 3,
  "ClassType": "Application Class",
  "ClassAttributes": [
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP010#001"
    },
    {
      "Type": "Property",
      "Reference": "TDOC#02-PRP011#001"
    }
  ]
```

```
    }
  ]
}
```

The first two classes, **CLS002** and **CLS003**, are the property block classes and the third class, **CLS004**, is the application class that references the two property blocks.

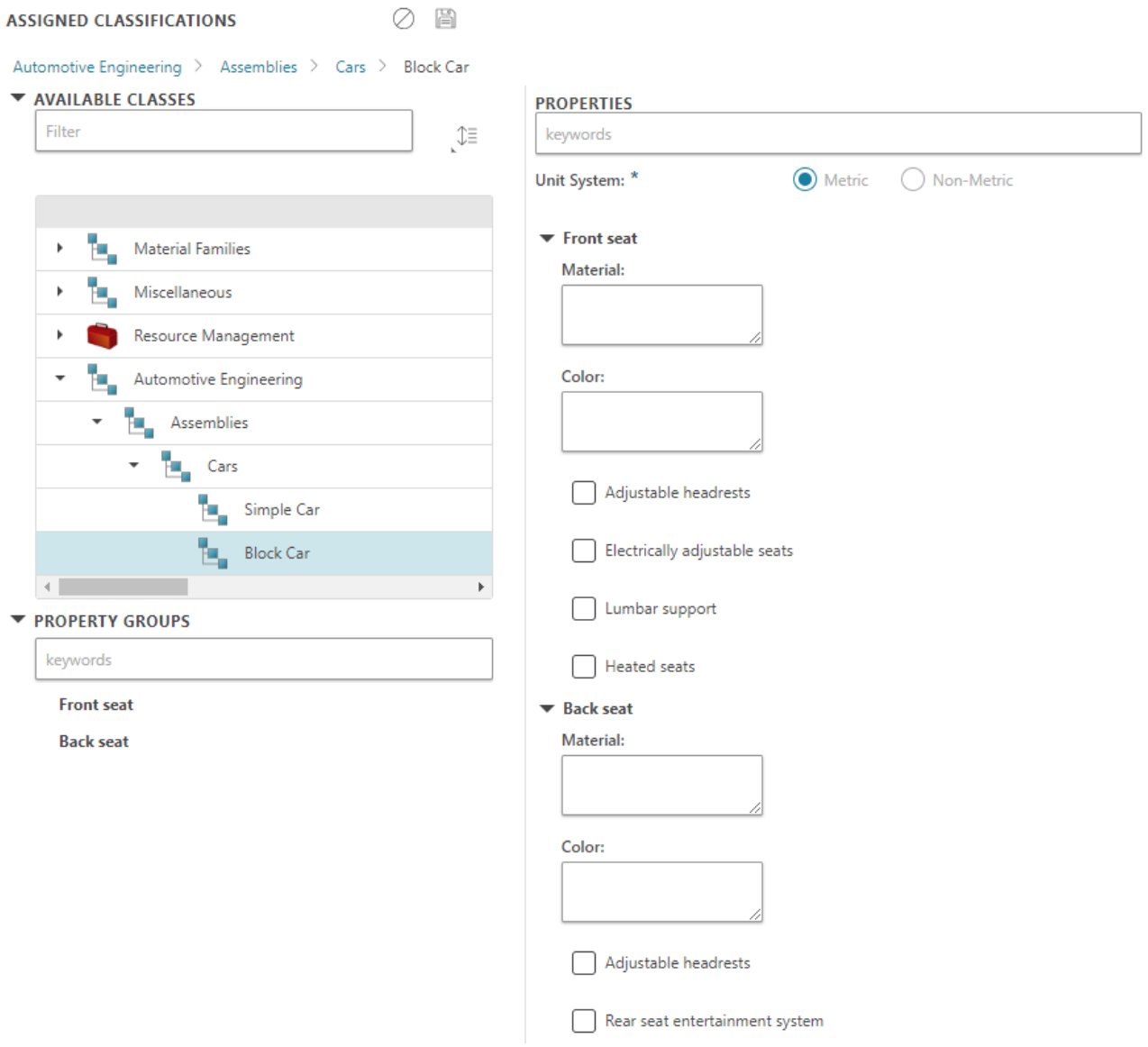
3. Import the node definition that makes the class visible in Active Workspace.

   The **Block car** node must be displayed in the **Cars** node that you created in the first example. To do this, import the new node specifying the **Car** node, **CSTAA1**, as the parent. Save the following in a file named **blockNodeDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "NodeDefinitions": [
    {
      "Namespace": "TDOC",
      "ID": "CSTAA3",
      "Revision": "001",
      "Parent": {
        "Namespace": "TDOC",
        "ID": "CSTAA1",
        "Revision": "001"
      },
      "Name": "Block Car",
      "ApplicationClass": {
        "Namespace": "TDOC",
        "ID": "CLS004",
        "Revision": "001"
      }
    }
  ]
}
```

4. Import the definitions into the database using **Classification Manager** or with **clsutility commands**.

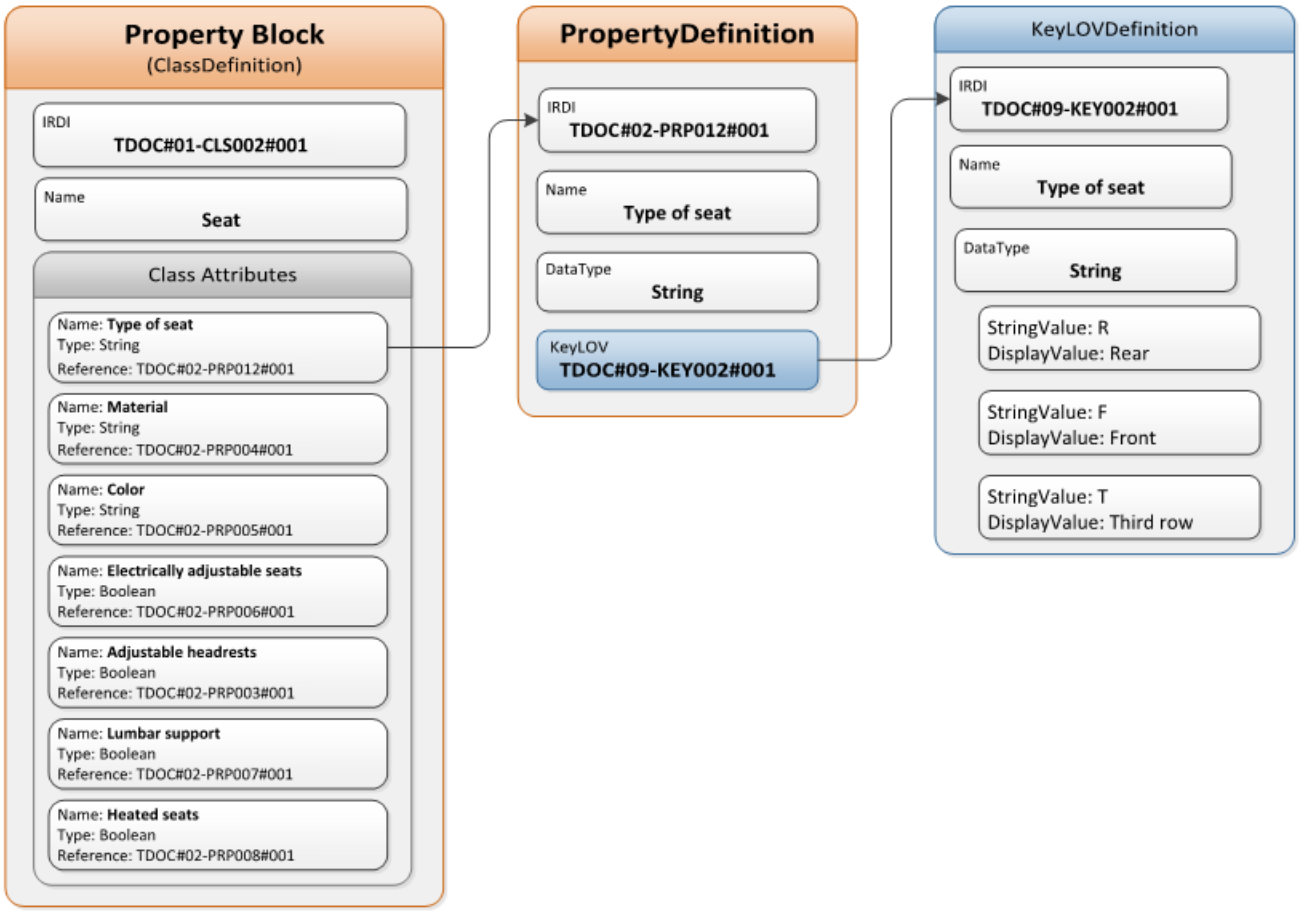
The structure is displayed as follows:



Import a class with cardinal properties

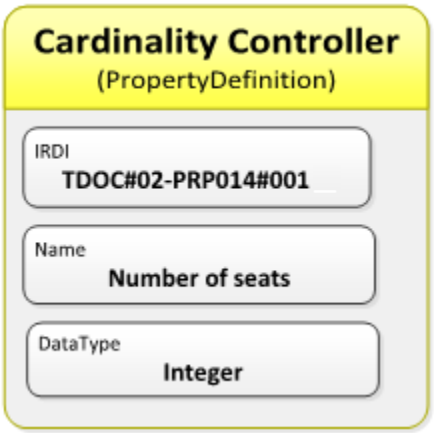
This example shows how to create a class where we can specify the number of seat rows in a car during classification. For example, when classifying a sports car, there is only one row of seats, so only one set of seat properties is necessary. When classifying a family van, there are three rows of seats that each need a set of properties to describe them.

We begin with a set of properties, a property block:

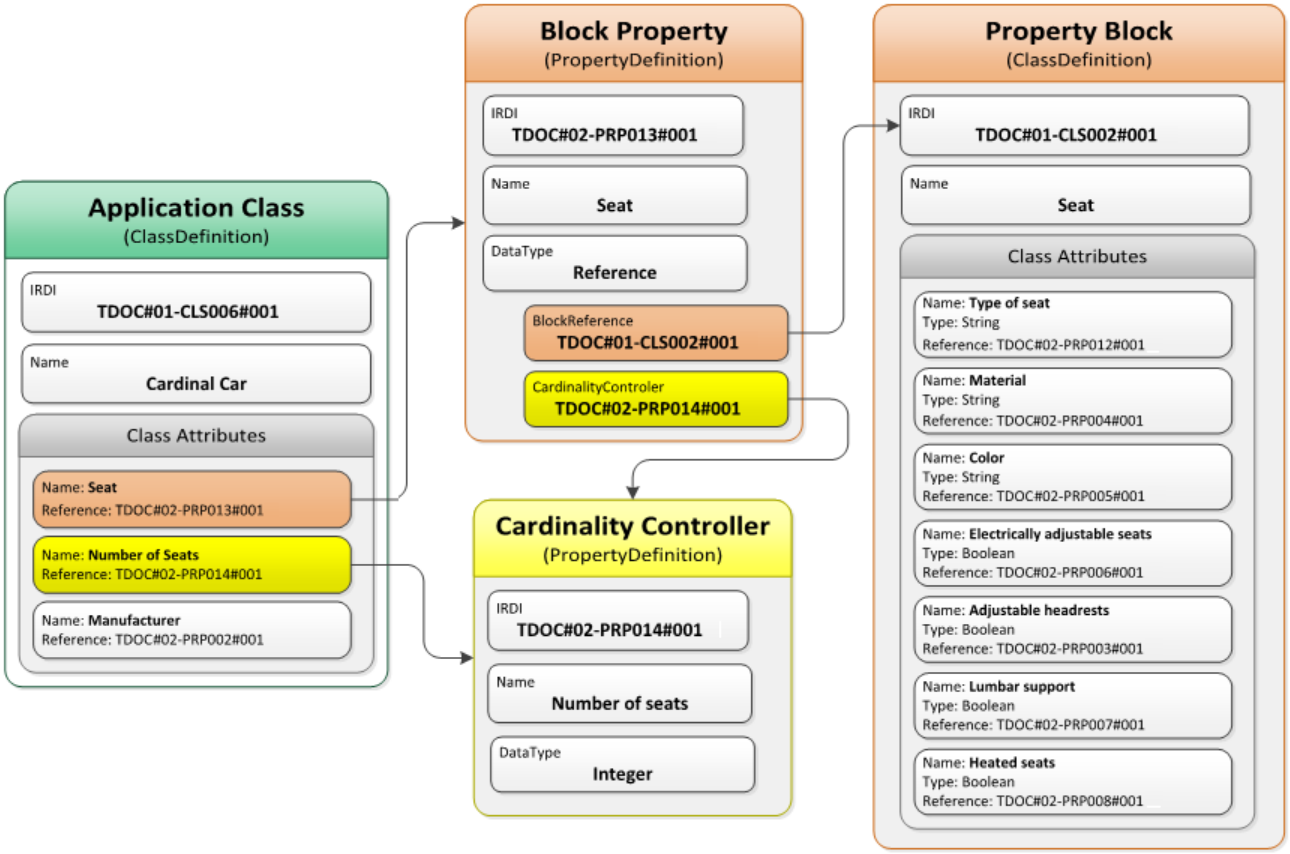


This property block class contains the same attributes as in the previous example, but additionally includes a new key-LOV property, **Type of seat**, where you can specify whether the seats are front, back, or third-row seats.

The number of times that this block of properties is displayed in the user interface is determined by a special property called a *cardinality controller*:



A cardinality controller is referenced by another property and is only of significance if set to **true**. If this is the case, then the user can specify the number of times that the set of properties is displayed.



To create the JSON files required to import these classes and properties:

1. Create the key-LOV definitions.

   This example introduces a new key-LOV, **Type of seat**, with three values, **Front**, **Rear**, and **Third**. Save the following in a file named **cardinalKeyLOVDef.json**

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "KeyLOVDefinitions": [
    {
      "ObjectType": "09",
      "Namespace": "TDOC",
      "ID": "KEY002",
      "Revision": "001",
      "Name": "Type of seat",
      "Status": "Develop",

      "LOVItems": {
```

```
        "DataType": "String",
        "LOVStringItems": [
          {
            "StringValue": "F",
            "DisplayValue": "Front"
          },
          {
            "StringValue": "R",
            "DisplayValue": "Rear"
          },
          {
            "StringValue": "T",
            "DisplayValue": "Third row"
          }
        ]
      }
    }
  ]
}
```

2. Create the individual property definitions.

   There are three new properties in this example: **Type of seat** that references the new key-LOV, **Seat**, the property that specifies the property block class that is to be displayed and acts as an indicator that cardinality is involved, and the **Number of seats** that holds the count.

   Save the following in a file named **cardinalPropDef.json**:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "PropertyDefinitions": [
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP012",
      "Revision": "001",
      "Name": "Type of seat",
      "Status": "Develop",
      "DataType": {
        "Type": "String",
        "KeyLOV": "TDOC#09-KEY002#001"
      }
    },
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP013",
```

```
      "Revision": "001",
      "Name": "Seat",
      "Status": "Develop",
      "DataType": {
        "Type": "Reference",
        "BlockReference": "TDOC#01-CLS005#001",
        "CardinalityController": "TDOC#02-PRP014#001"
      }
    },
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP014",
      "Revision": "001",
      "Name": "Number of seats",
      "Status": "Develop",
      "DataType": {
        "Type": "Integer"
      }
    }
  ]
}
```

3. Create the class definitions, including the property block classes.

   This example introduces two new classes, **CLS005**, the property block class, and **CLS006**, the application class. Save the following in a file named **cardinalClassDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS005",
      "Revision": "001",
      "Name": "Seat properties",
      "Status": "Develop",
      "IsAbstract": true,
      "UnitSystem": 3,
      "ClassType": "Block",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP012#001"
        },
        {
          "Type": "Property",
```

```
        "Reference": "TDOC#02-PRP004#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP005#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP006#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP003#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP007#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP008#001"
      }
    ]
  },
  {
    "ObjectType": "01",
    "Namespace": "TDOC",
    "ID": "CLS006",
    "Revision": "001",
    "Name": "Cardinal car",
    "Status": "Develop",
    "IsAbstract": false,
    "UnitSystem": 3,
    "ClassType": "Application Class",
    "ClassAttributes": [
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP013#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP014#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP002#001"
      }
    ]
  }
```

```
    ]
}
```

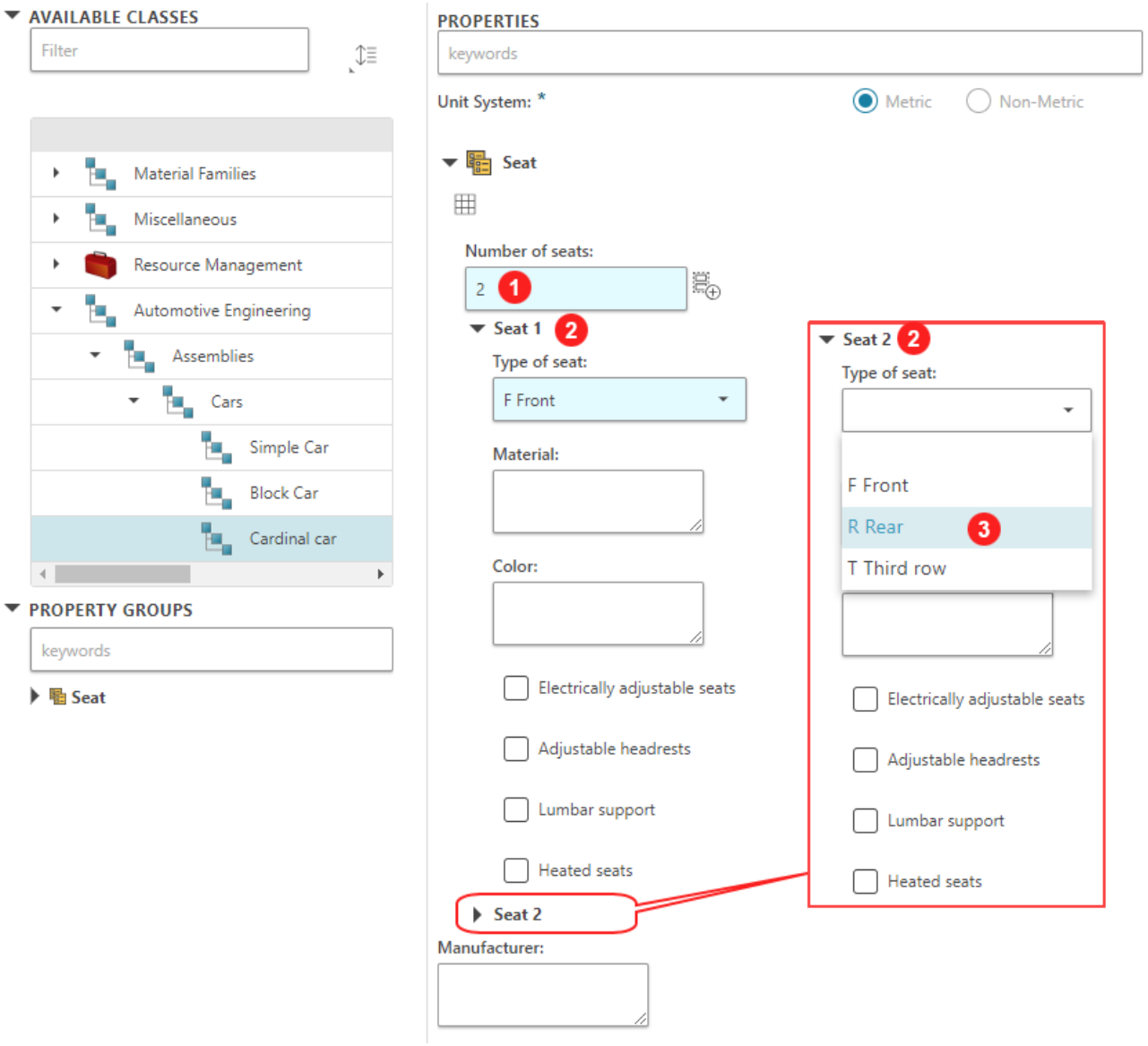
4. Import the node definition that makes the class visible in.

   The **Cardinal car** node must be displayed in the **Cars** node that we created in the first example. To do this, import the new node specifying the **Car** node, **CSTAA1**, as the parent. Save the following in a file named **cardinalNodeDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "NodeDefinitions": [
    {
      "Namespace": "TDOC",
      "ID": "CSTAA4",
      "Revision": "001",
      "Parent": {
        "Namespace": "TDOC",
        "ID": "CSTAA1",
        "Revision": "001"
      },
      "Name": "Cardinal car",
      "ApplicationClass": {
        "Namespace": "TDOC",
        "ID": "CLS006",
        "Revision": "001"
      }
    }
  ]
}
```

5. Import the definitions into the database using **Classification Manager** or with **clsutility commands**.

The structure is displayed as follows:



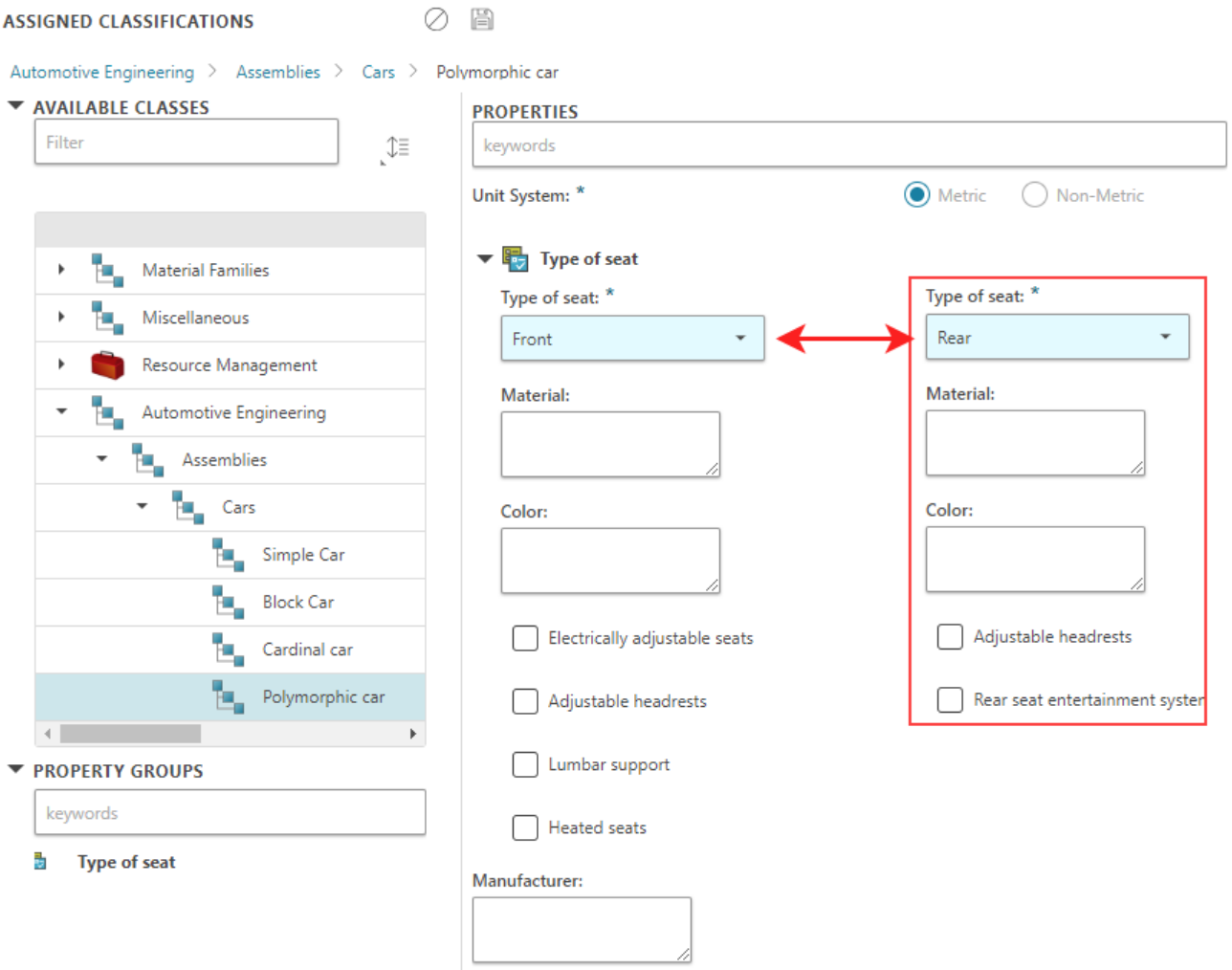
After specifying the number of seats in 1, that number of **Seat** property blocks is displayed 2. You can assign different values for each seat row. With the new key-LOV 3, you can specify a separate type of seat for each row.

Import a class with polymorphic properties

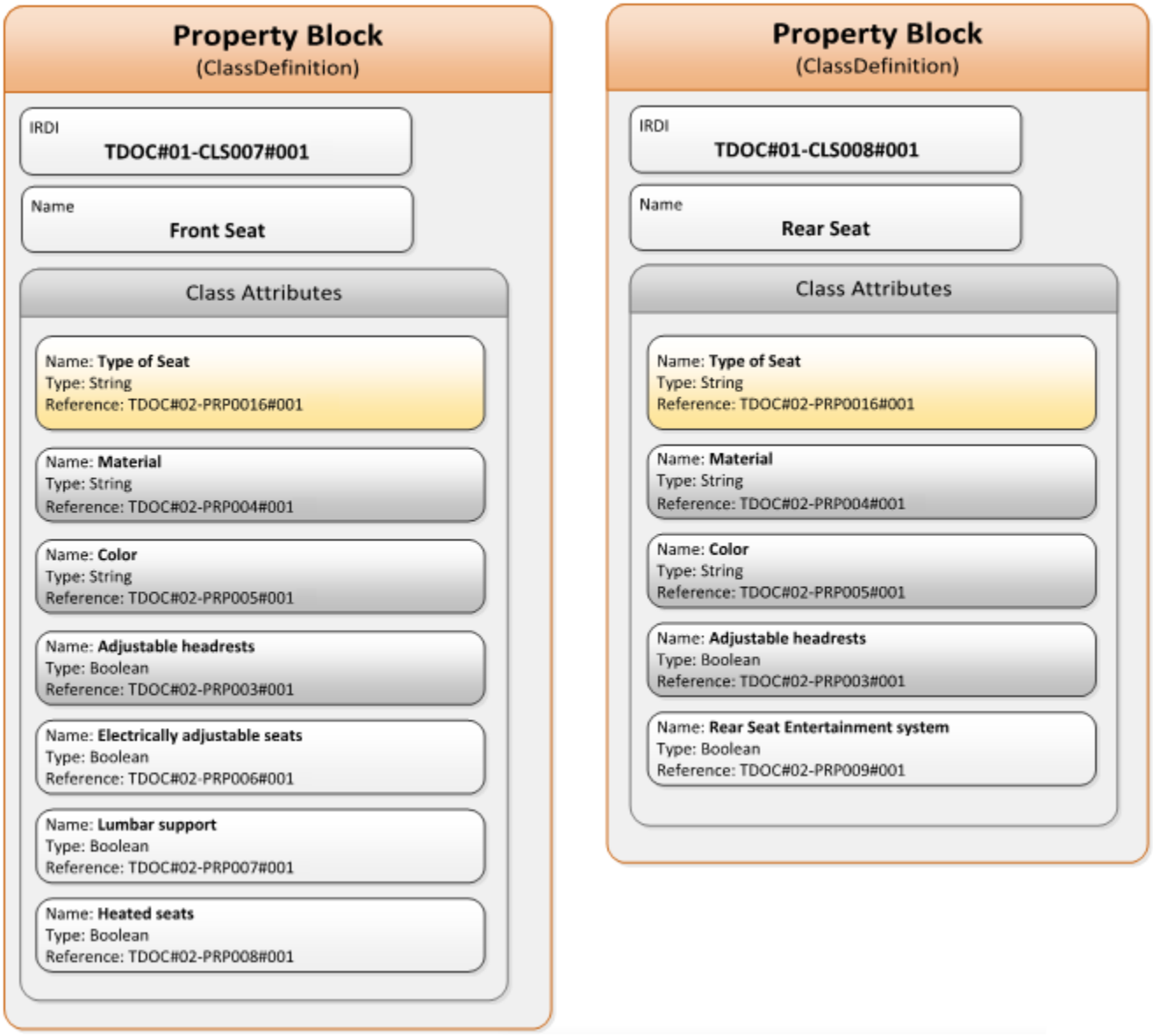
This example shows how to create a class where we can specify the properties displayed depending on the type of seat row in a car that you select during classification. The following table displays the different properties available for each seat row type:

Front seat	Rear seat
Material	Material
Color	Color
Adjustable headrests	Adjustable headrests
Electrically adjustable seats	Rear seat entertainment
Lumbar support	
Heated seats	

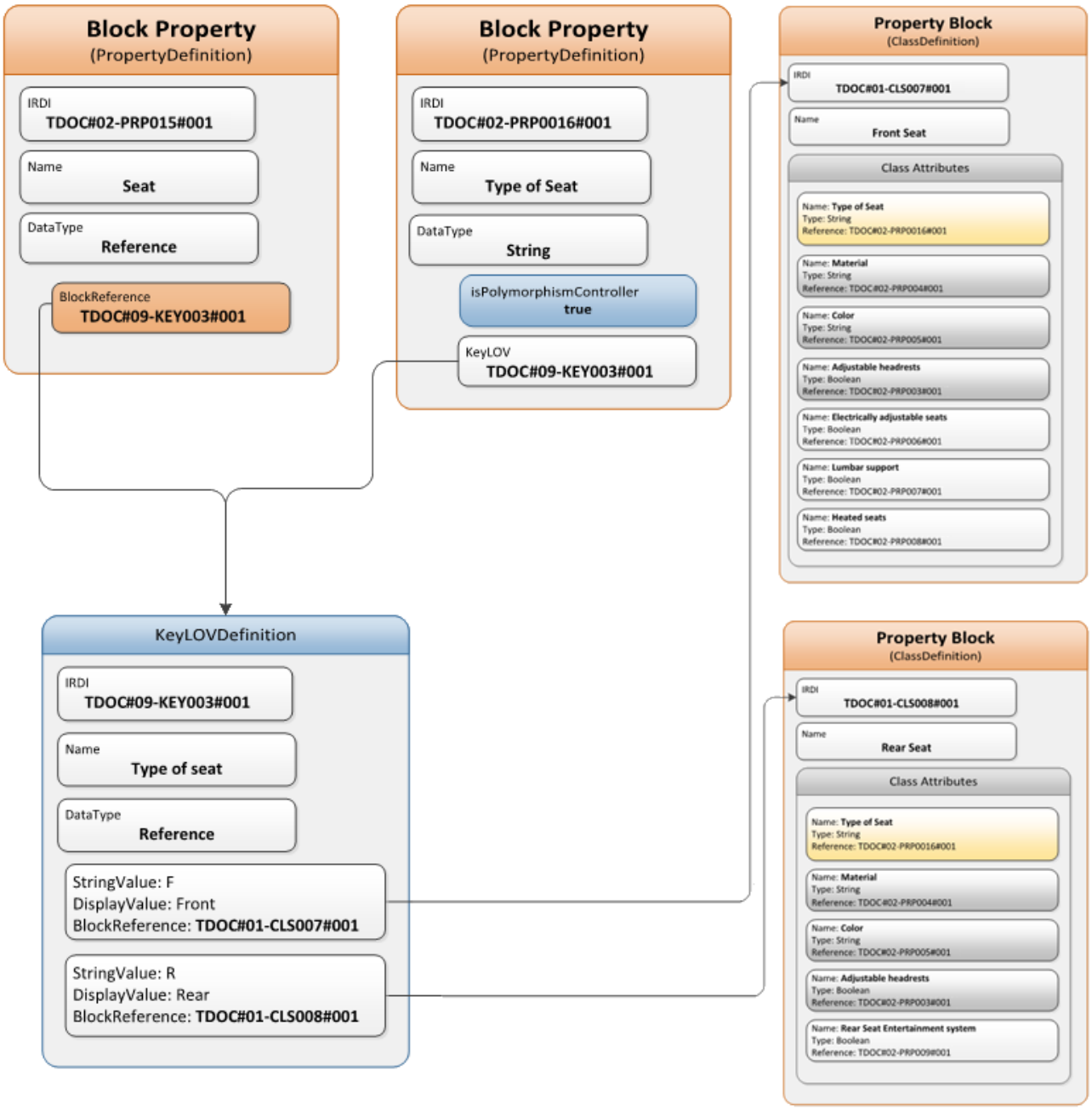
These are the same properties used in the **second example**. The difference is that in this example, you can specify the seat type during classification and, depending on this selection, display different properties. The concept of displaying different properties depending on the value of an additional property is referred to as *polymorphism*.



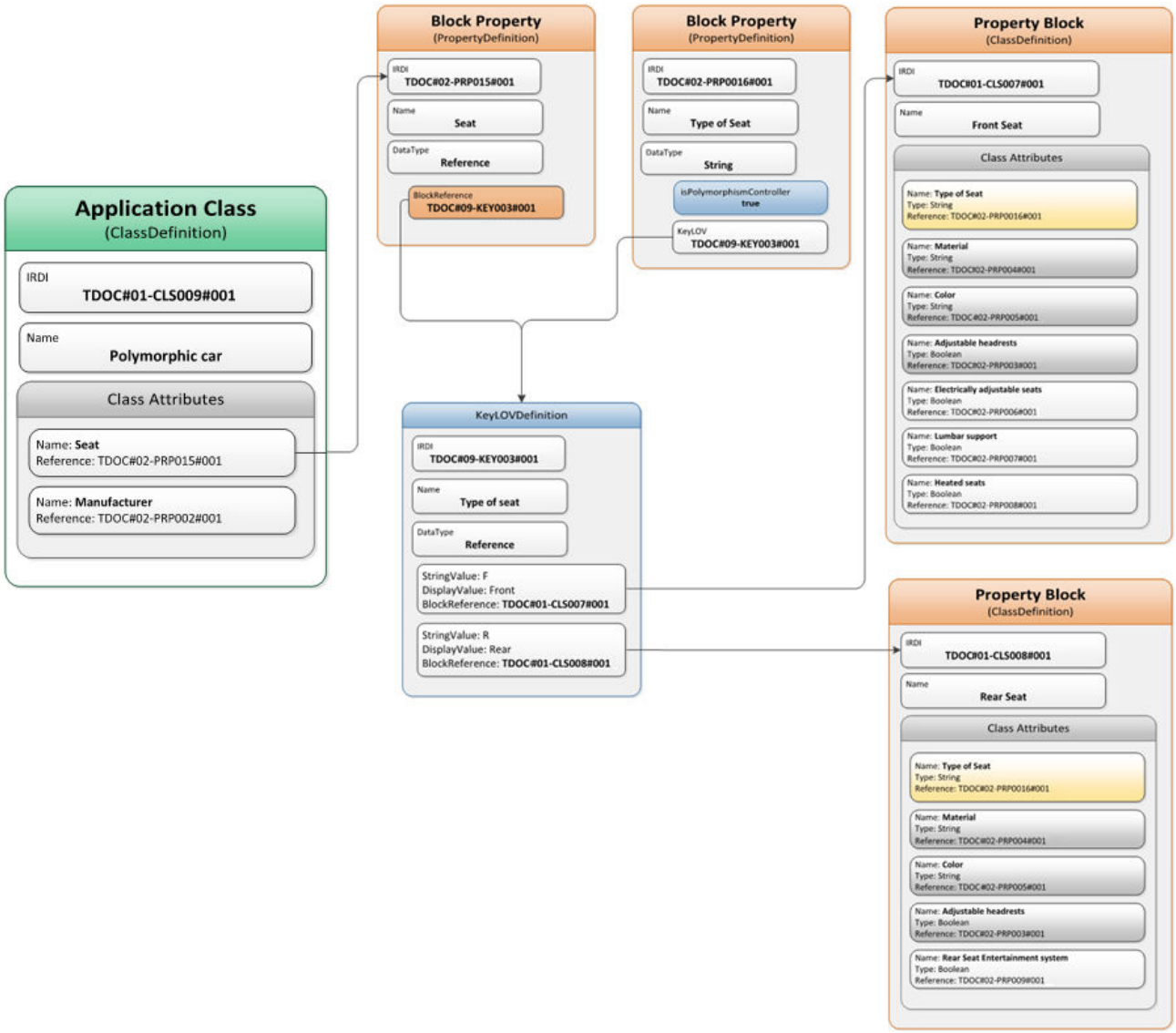
This example contains two classes describing the front and rear seats:



Each of the property block classes contains a special property that specifies the type of seat. This property references a key-LOV with the values **Front** or **Rear**.



The following application class contains the polymorphic **Seat** property that references the key-LOV.



Another property that references the key-LOV, **PRP016**, contains the **isPolymorphismController** attribute. When this property is used in a property block class (**CLS0007** or **CLS008**), the **isPolymorphismController** attribute indicates that the property is polymorphic and that its value determines the list of class properties displayed in the user interface.

To create the JSON files for this example:

1. Create the key-LOV definition.

   This example introduces one new key-LOV that is saved in a file called **polymorphicKeylovDef.json**:

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
```

```
  "KeyLOVDefinitions": [
    {
      "ObjectType": "09",
      "Namespace": "TDOC",
      "ID": "KEY003",
      "Revision": "001",
      "Name": "Type of seat",
      "Status": "Develop",
      "LOVItems": {
        "DataType": "Reference",
        "LOVReferenceItems": [
          {
            "StringValue": "F",
            "DisplayValue": "Front",
            "BlockReference": "TDOC#01-CLS007#001"
          },
          {
            "StringValue": "R",
            "DisplayValue": "Rear",
            "BlockReference": "TDOC#01-CLS008#001"
          }
        ]
      }
    }
  ]
}
```

The values of the key-LOV, **Front** and **Rear**, reference their respective property block classes, **CLS007** and **CLS008**.

2. Create the individual property definitions.

   This example uses many of the properties created in the earlier examples, as well as two new properties. Create the following JSON file and save it in a file named **polymorphicPropDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "PropertyDefinitions": [
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP015",
      "Revision": "001",
      "Name": "Seat",
      "Status": "Develop",
      "DataType": {
        "Type": "Reference",
```

```
        "BlockReference": "TDOC#09-KEY003#001"
      }
    },
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP016",
      "Revision": "001",
      "Name": "Type of Seat",
      "Status": "Develop",
      "DataType": {
        "Type": "String",
        "KeyLOV": "TDOC#09-KEY003#001",
        "IsPolymorphismController": true
      }
    }
  ]
}
```

PRP016 contains the **IsPolymorphismController** attribute.

3. Create the class definitions, including the property block classes.

   This example introduces two new property block classes very similar to those in the block class example, but additionally containing the polymorphic **PRP016** property. Save this in a file named **polymorphicClassDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS007",
      "Revision": "001",
      "Name": "Front seat",
      "Status": "Develop",
      "IsAbstract": true,
      "UnitSystem": 3,
      "ClassType": "Block",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP016#001"
        },
        {
          "Type": "Property",
```

```
        "Reference": "TDOC#02-PRP004#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP005#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP006#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP003#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP007#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP008#001"
      }
    ]
  },
  {
    "ObjectType": "01",
    "Namespace": "TDOC",
    "ID": "CLS008",
    "Revision": "001",
    "Name": "Back seat",
    "Status": "Develop",
    "IsAbstract": true,
    "UnitSystem": 3,
    "ClassType": "Block",
    "ClassAttributes": [
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP016#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP004#001"
      },
      {
        "Type": "Property",
        "Reference": "TDOC#02-PRP005#001"
      },
      {
        "Type": "Property",
```

```
            "Reference": "TDOC#02-PRP003#001"
          },
          {
            "Type": "Property",
            "Reference": "TDOC#02-PRP009#001"
          }
        ]
      },
      {
        "ObjectType": "01",
        "Namespace": "TDOC",
        "ID": "CLS009",
        "Revision": "001",
        "Name": "Polymorphic car",
        "Status": "Develop",
        "IsAbstract": false,
        "UnitSystem": 3,
        "ClassType": "Application Class",
        "ClassAttributes": [
          {
            "Type": "Property",
            "Reference": "TDOC#02-PRP015#001"
          },
          {
            "Type": "Property",
            "Reference": "TDOC#02-PRP002#001"
          }
        ]
      }
    ]
}
```

The application class **CLS009** references **PRP015**, which is associated with the polymorphic attribute **PRP016** through the key-LOV.

4. Import the node definition that makes the class visible.

   The **Polymorphic car** node must be displayed in the **Cars** node that we created in the first example. To do this, import the new node specifying the **Car** node **CSTAA1** as the parent. Save the following in a file named **polymorphicNodeDef.json**.

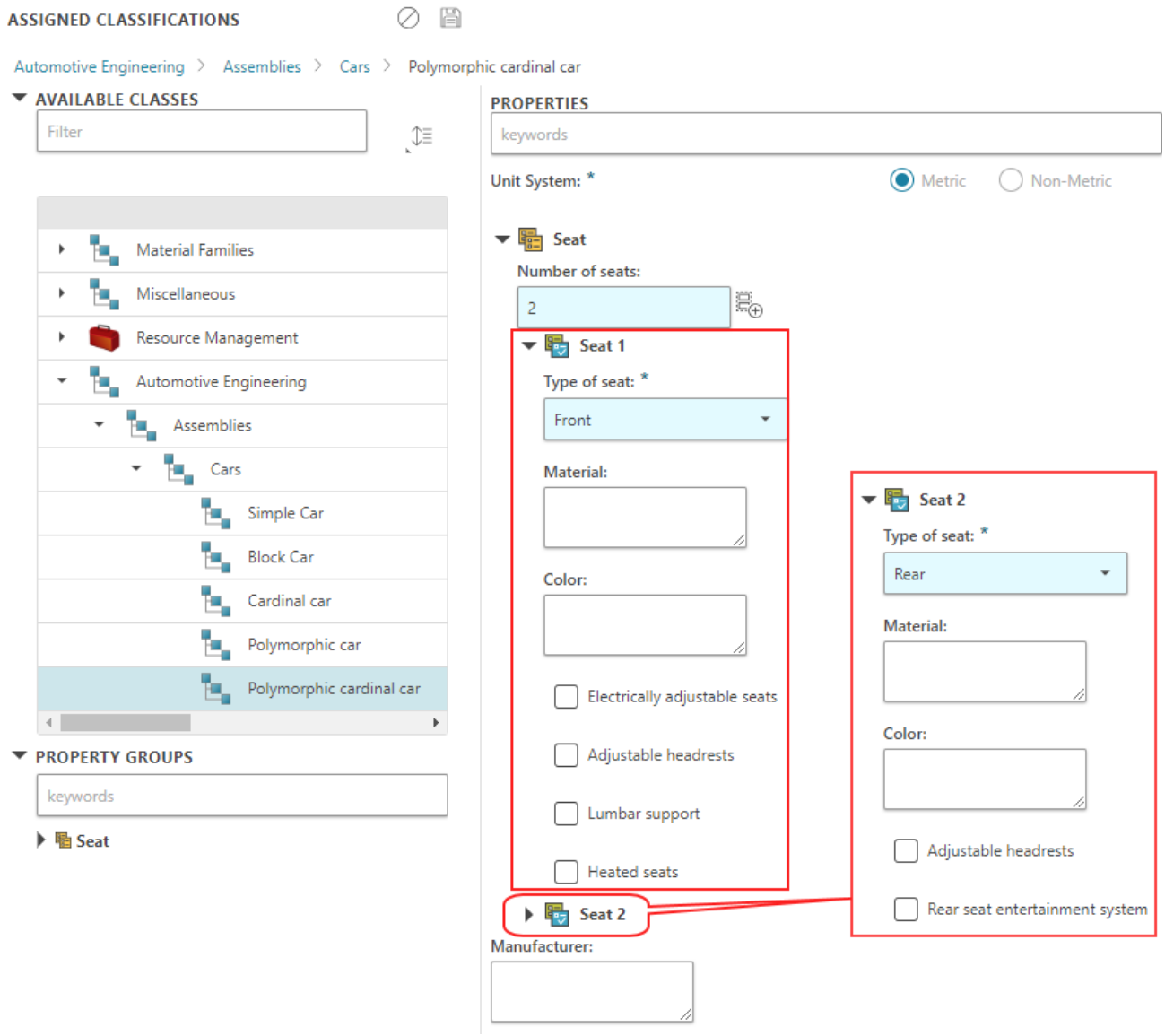
```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "NodeDefinitions": [
    {
      "Namespace": "TDOC",
      "ID": "CSTAA5",
```

```
        "Revision": "001",
        "Parent": {
          "Namespace": "TDOC",
          "ID": "CSTAA1",
          "Revision": "001"
        },
        "Name": "Polymorphic car",
        "ApplicationClass": {
          "Namespace": "TDOC",
          "ID": "CLS009",
          "Revision": "001"
        }
      }
    ]
  }
```

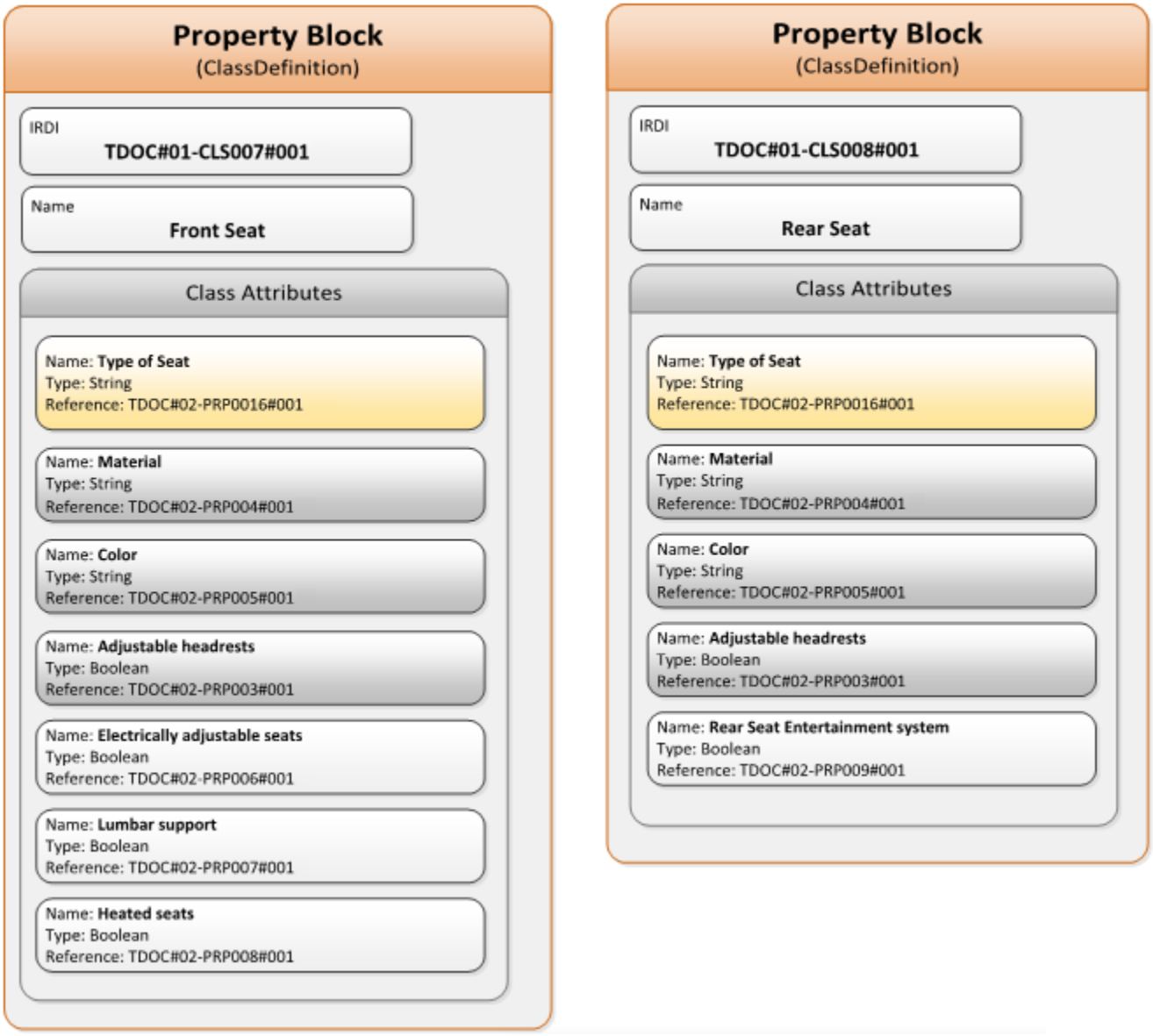
5. Import the definitions into the database using **Classification Manager** or with **clsutility commands**.

Import a class with polymorphic and cardinal properties

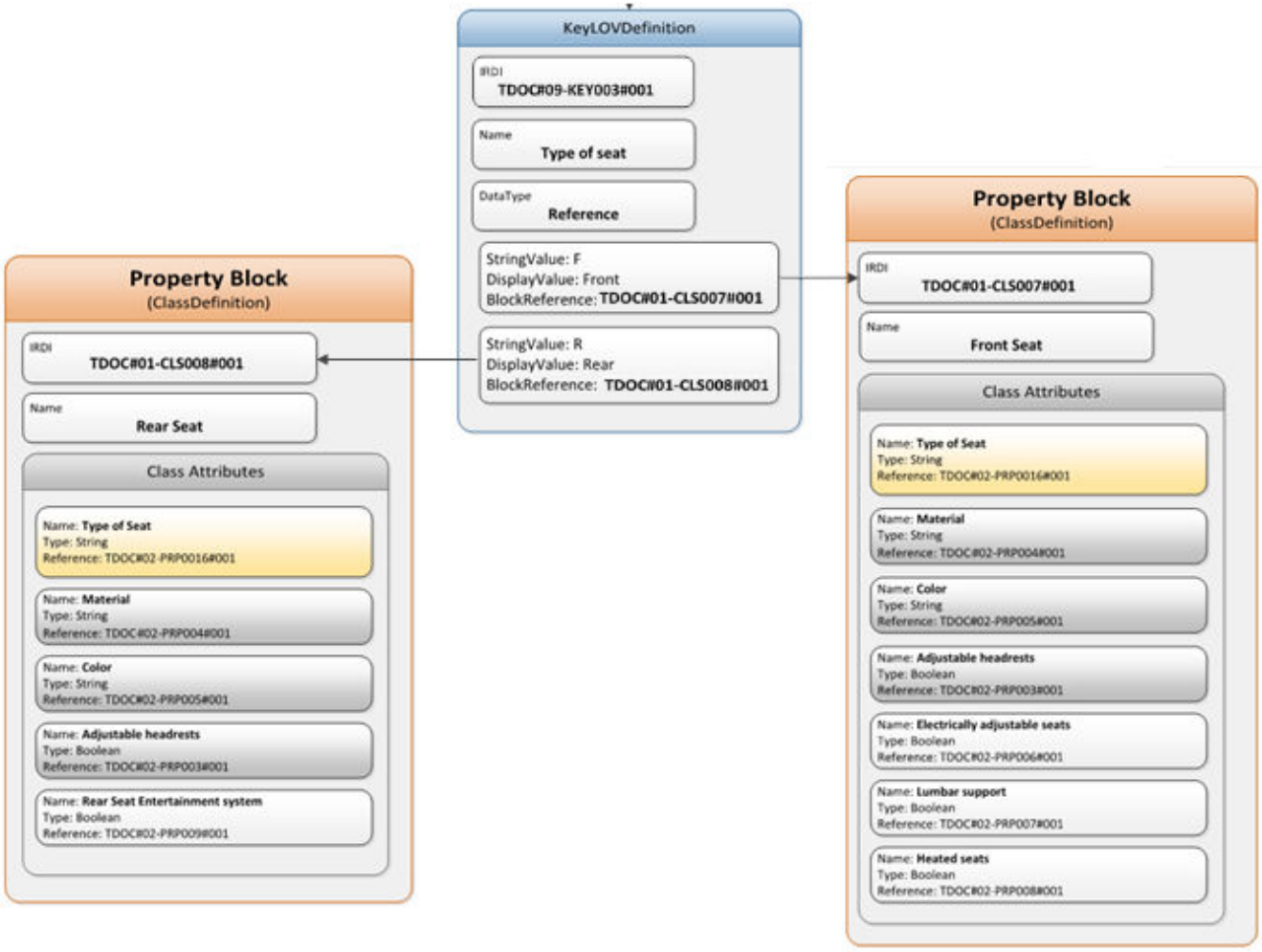
This example shows how to create a class where you can first specify the number of seat rows a car has during classification. For each seat row, the properties displayed depend on the type of seat row that you select.



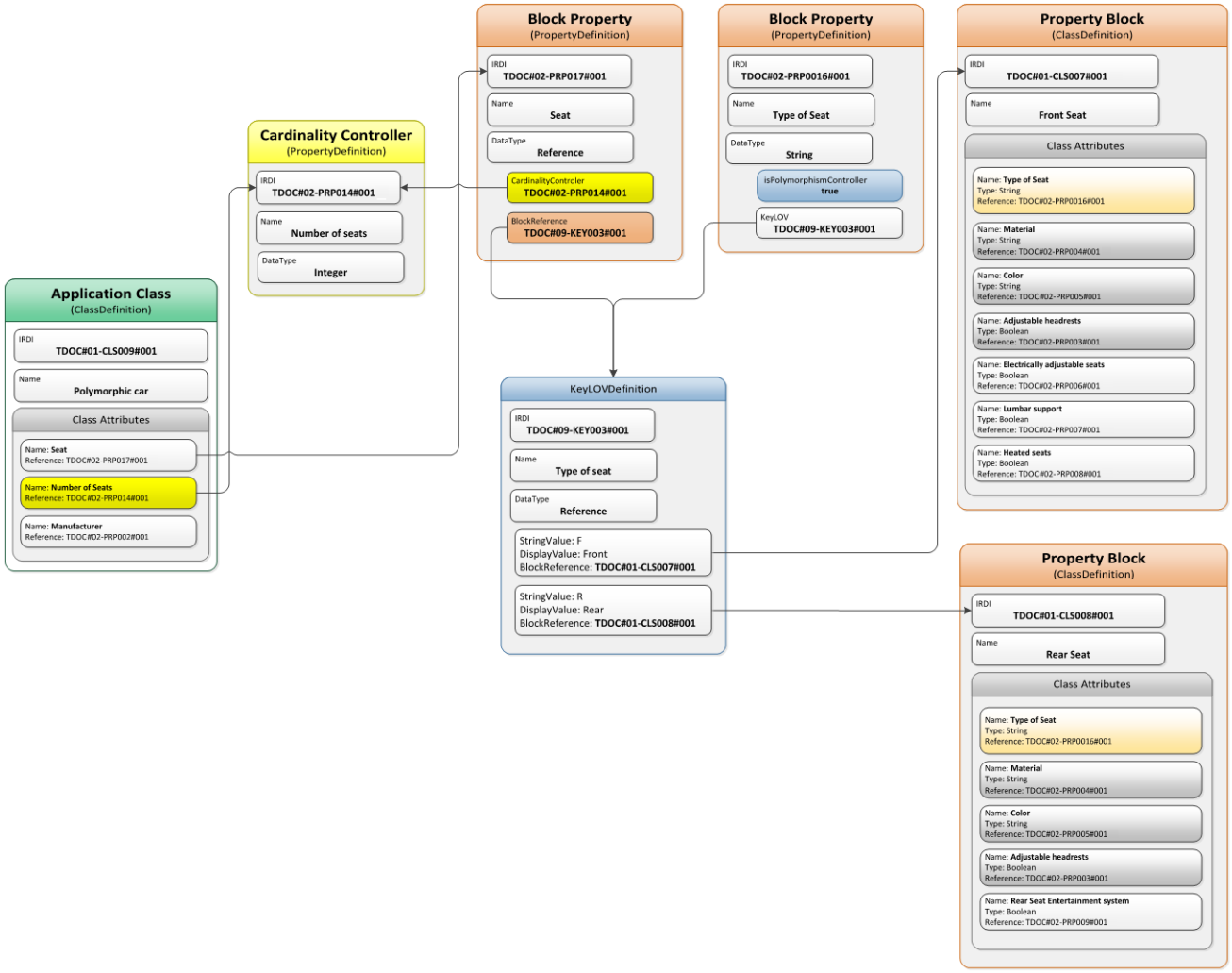
This example uses the same two property block classes as the **previous example**, describing the front and rear seats:



Again, these classes are referenced by the same key-LOV as in the previous example, which determines which of these classes is displayed:



The application class references the properties that determine the polymorphism and cardinality:



After completing the previous examples, there are only a few new objects required to import this example. To create the JSON files for this example:

1. Create the individual property definitions.

   This example uses **PRP014** created in the cardinality example and **PRP016** created in the polymorphism example. One new property is required that references the key-LOV created in the polymorphism example. Create the following JSON file and save it in a file named **poly_cardPropDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "PropertyDefinitions": [
    {
      "ObjectType": "02",
      "Namespace": "TDOC",
      "ID": "PRP017",
```

```
        "Revision": "001",
        "Name": "Seat",
        "Status": "Develop",
        "DataType": {
          "Type": "Reference",
          "BlockReference": "TDOC#09-KEY003#001",
          "CardinalityController": "TDOC#02-PRP014#001"
        }
      }
    ]
  }
```

This property references the key-LOV definition, as well as the cardinality controller property. The polymorphism is achieved by the reference of **PRP016** to the key-LOV.

2. Create the class definitions.

   This example introduces only one new application class. Save this in a file named **poly_cardClassDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "ClassDefinitions": [
    {
      "ObjectType": "01",
      "Namespace": "TDOC",
      "ID": "CLS010",
      "Revision": "001",
      "Name": "Polymorphic cardinal car",
      "Status": "Develop",
      "IsAbstract": false,
      "UnitSystem": 3,
      "ClassType": "Application Class",
      "ClassAttributes": [
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP014#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP017#001"
        },
        {
          "Type": "Property",
          "Reference": "TDOC#02-PRP002#001"
        }
      ]
```

```
      }
    ]
  }
```

3. Import the node definition that makes the class visible.

   The **Polymorphic cardinal car** node must be displayed in the **Cars** node that we created in the first example. To do this, import the new node specifying the **Car** node **CSTAA1** the as parent. Save the following in a file named **poly_cardNodeDef.json**.

```
{
  "SchemaVersion": "1.0.0",
  "Locale": "en_US",
  "NodeDefinitions": [
    {
      "Namespace": "TDOC",
      "ID": "CSTAA6",
      "Revision": "001",
      "Parent": {
        "Namespace": "TDOC",
        "ID": "CSTAA1",
        "Revision": "001"
      },
      "Name": "Polymorphic cardinal car",
      "ApplicationClass": {
        "Namespace": "TDOC",
        "ID": "CLS010",
        "Revision": "001"
      }
    }
  ]
}
```

4. Import the definitions into the database using **Classification Manager** or with **clsutility commands**.

Importing classification objects to CST

Classification objects can be imported using the JSON format. The schema file for this is:

..\TD\classification\json\schema\advanced\Classification_Save_PropertyRecords_Request_advanced.schema.json

When creating the JSON file, you can specify whether only the item for the property record is created, or additionally specify a class into which the item is immediately classified during import. Importing the following example classifies an item into the **Polymorphic cardinal car** class that was created in the **import example**.

Tip:

Find the example as text that you can copy **here**.

```
{
    "SchemaVersion": "1.0.0",
    "Locale": "en_US",
    "PropertyRecords": [
        {
            "ID": "029159/A",
            "ObjectType": "PR",
            "ClassDefinition": "TDOC#01-CLS010#001",
            "UnitSystem": 1,
            "ClassifiedObject": {
                "ObjectType": "Item",
                "ClassifyRevision": true,
                "ID": [
                    {
                        "PropertyName": "item_id",
                        "PropertyValue": "029159"
                    }
                ],
                "Properties": [
                    {
                        "PropertyName": "object_name",
                        "PropertyValue": "my_car"
                    }
                ]
            },
            "Properties": [
                {
                    "ID": "TDOC#02-PRP014#001",
                    "Name": "Number of seats",
                    "Value": 2
                },
                {
                    "ID": "TDOC#02-PRP017#001",
                    "Name": "Seat",
                    "Value": [
                        {
                            "Index": 1,
                            "ClassDefinition": "TDOC#01-CLS008#001",
                            "Properties": [
                                {
                                    "ID": "TDOC#02-PRP004#001",
                                    "Name": "Material",
                                    "Value": "Polyester"
                                },
                                {
                                    "ID": "TDOC#02-PRP005#001",
                                    "Name": "Color",
                                    "Value": "Black"
                                },
                                {
                                    "ID": "TDOC#02-PRP003#001",
                                    "Name": "Adjustable headrests",
                                    "Value": true
                                },
                                {
                                    "ID": "TDOC#02-PRP009#001",
                                    "Name": "Rear seat entertainment system",
                                    "Value": true
                                }
                            ]
                        }
                    ]
                },
                {
                    "ID": "TDOC#02-PRP002#001",
                    "Name": "Manufacturer",
                    "Value": "Fancy Car Company"
                }
            ]
        }
    ]
}
```

In the JSON file:

- Lines 10-25 create the classified item with ID **029159** and name **my_car**.
- Line 30 creates 2 instances (a cardinality of 2) of **Number of seats** and these seats are described in lines 36-99.
- The index shown in line 37 and 73 introduces each of the new seat instances.
- Lines 39 and 75 begin the list of properties for each type of seat.
- The list of properties specific to each seat type is described in lines 40-71 and lines

The **Name** property is not mandatory and is added to this example for purposes of clarity.

To import this property record that is saved in a file called **poly_car_prop_rec.json**, you must use the **clsutility**:

```
clsutility -create -classification_objects
-request="poly_car_prop_rec.json"
```

JSON classification object input example

```
{
    "SchemaVersion": "1.0.0",
    "Locale": "en_US",
    "PropertyRecords": [
        {
            "ID": "029159/A",
            "ObjectType": "PR",
            "ClassDefinition": "TDOC#01-CLS010#001",
            "UnitSystem": 1,
            "ClassifiedObject": {
                "ObjectType": "Item",
                "ClassifyRevision": true,
                "ID": [
                    {
                        "PropertyName": "item_id",
                        "PropertyValue": "029159"
                    }
                ],
                "Properties": [
                    {
                        "PropertyName": "object_name",
                        "PropertyValue": "my_car"
                    }
                ]
            },
            "Properties": [
                {
                    "ID": "TDOC#02-PRP014#001",
                    "Name": "Number of seats",
                    "Value": 2
```

```
},
{
    "ID": "TDOC#02-PRP017#001",
    "Name": "Seat",
    "Value": [
        {
            "Index": 1,
            "ClassDefinition": "TDOC#01-CLS007#001",
            "Properties": [
                {
                    "ID": "TDOC#02-PRP004#001",
                    "Name": "Material",
                    "Value": "Leather"
                },
                {
                    "ID": "TDOC#02-PRP005#001",
                    "Name": "Color",
                    "Value": "Red"
                },
                {
                    "ID": "TDOC#02-PRP006#001",
                    "Name": "Electrically adjustable seats",
                    "Value": true
                },
                {
                    "ID": "TDOC#02-PRP003#001",
                    "Name": "Adjustable headrests",
                    "Value": true
                },
                {
                    "ID": "TDOC#02-PRP007#001",
                    "Name": "Lumbar support",
                    "Value": true
                },
                {
                    "ID": "TDOC#02-PRP008#001",
                    "Name": "Heated seats",
                    "Value": true
                }
            ]
        },
        {
            "Index": 2,
            "ClassDefinition": "TDOC#01-CLS008#001",
            "Properties": [
                {
                    "ID": "TDOC#02-PRP004#001",
                    "Name": "Material",
                    "Value": "Polyester"
                },
                {
                    "ID": "TDOC#02-PRP005#001",
                    "Name": "Color",
                    "Value": "Black"
                },
                {
                    "ID": "TDOC#02-PRP003#001",
                    "Name": "Adjustable headrests",
                    "Value": true
                },
```

```
                                {
                                    "ID": "TDOC#02-PRP009#001",
                                    "Name": "Rear seat entertainment system",
                                    "Value": true
                                }
                            ]
                        }
                    ]
                },
                {
                    "ID": "TDOC#02-PRP002#001",
                    "Name": "Manufacturer",
                    "Value": "Fancy Car Company"
                }
            ]
        }
    ]
}
```

clsutility commands required to import example json files

When importing files for **this example**, if you choose to import using the **clsutility**, run each of the following commands in a Teamcenter command window:

Import a class with simple properties

```
clsutility -u=user -p=password -g=group -create -keylov_definitions
-request="simpleKeylovDef.json"
clsutility -u=user -p=password -g=group -create -property_definitions
-request="simplePropDef.json"
clsutility -u=user -p=password -g=group -create -class_definitions
-request="simpleApplicationClassDef.json"
clsutility -u=user -p=password -g=group -create -node_definitions
-request="simpleNodeDef.json"
```

Import a class with a block properties

```
clsutility -u=user -p=password -g=group -create -property_definitions
-request="blockPropDef.json"
clsutility -u=user -p=password -g=group -create -class_definitions
-request="blockClassDef.json"
clsutility -u=user -p=password -g=group -create -node_definitions
-request="blockNodeDef.json"
```

Import a class with cardinal properties

```
clsutility -u=user -p=password -g=group -create -keylov_definitions
-request="cardinalKeyLOVDef.json"
clsutility -u=user -p=password -g=group -create -property_definitions
-request="cardinalPropDef.json"
clsutility -u=user -p=password -g=group -create -class_definitions
-request="cardinalClassDef.json"
clsutility -u=user -p=password -g=group -create -node_definitions
-request="cardinalNodeDef.json"
```

Import a class with polymorphic properties

```
clsutility -u=user -p=password -g=group -create -keylov_definitions
-request="polymorphicKeylovDef.json"
clsutility -u=user -p=password -g=group -create -property_definitions
-request="polymorphicPropDef.json"
clsutility -u=user -p=password -g=group -create -class_definitions
-request="polymorphicClassDef.json"
clsutility -u=user -p=password -g=group -create -node_definitions
-request="polymorphicNodeDef.json"
```

Import a class with polymorphic and cardinal properties

```
clsutility -u=user -p=password -g=group -create -property_definitions
-request="poly_cardPropDef.json"
clsutility -u=user -p=password -g=group -create -class_definitions
-request="poly_cardClassDef.json"
clsutility -u=user -p=password -g=group -create -node_definitions
-request="poly_cardNodeDef.json"
```

Import a property record

```
clsutility -create -classification_objects
-request="poly_car_prop_rec.json"
```

Export classified data for advanced classification

You can export classified data in the BMEcat XML from the user interface.

1. Open the classified data that you want to export.

2. Choose **More Commands** ••• > **Import/Export** > **Export Classification Data**.

3. In the **File Name** box, give a name to the file that will be downloaded.

4. Click **Export**.

When the export is complete, you are notified in the **Alert** area where you can view the details of the export.

The file is available in the **Target Object** section of the alert report. Download the file from that section.

This feature requires that Subscription Manager is running. If the Subscription Manager is not running, no notification is displayed in the **Alert** area, but the operation still occurs in the background.

Export classified objects using the clsutility command

To export the classification information (the *ICO*) for an object classified in a CST hierarchy, use the following **clsutility** command:

clsutility -u=*user* **-p=***password***-g=***group* **-find -classification_objects -request=***JSON_file* **-output=***path-to-output-file*

This command requires a JSON file resembling the following (see the Classification_FindRequest_advanced.schema.json schema file):

```
{
    "SchemaVersion": "1.0.0",
    "Locale" : "en_US",
    "ObjectIDList": [
        {
            "ID": "026144/A"
        }
    ]
}
```

The input JSON file contains a list of object IDs for which the ICO information is exported to a file of your choice.

Enable auditing for classification events

Overview of auditing classification events

You can allow audit logging of classification events as follows:

- **Make the audit logs visible for users.**

  You can specify who can view audit logs by updating the values of preferences.

- **Allow auditing for certain event types**

  By default, only the audit logs of modify and delete events are saved. You can activate audit logging for other classification events in the **Audit Definition** editor in Business Modeler IDE.

- **Allow auditing for class attributes**

  By default, the audit logs of class attributes are not saved. You can specify which class attributes can support auditing.

- **Test if you can view the audit logs**

Make audit logs visible for users

To make audit logs of Classification events visible for users, you must update certain preferences.

Procedure

1. Update the value of the **AWC_show_audit_logs** preference to **true**.

2. Update the value of the **AWC_show_classification_audit_log** preference to **true** for the group or role that needs access to the audit log.

Allow auditing for certain event types

By default, audit logs for modification and deletion events are saved. If you want to allow auditing for other event types, perform the following configurations.

Procedure

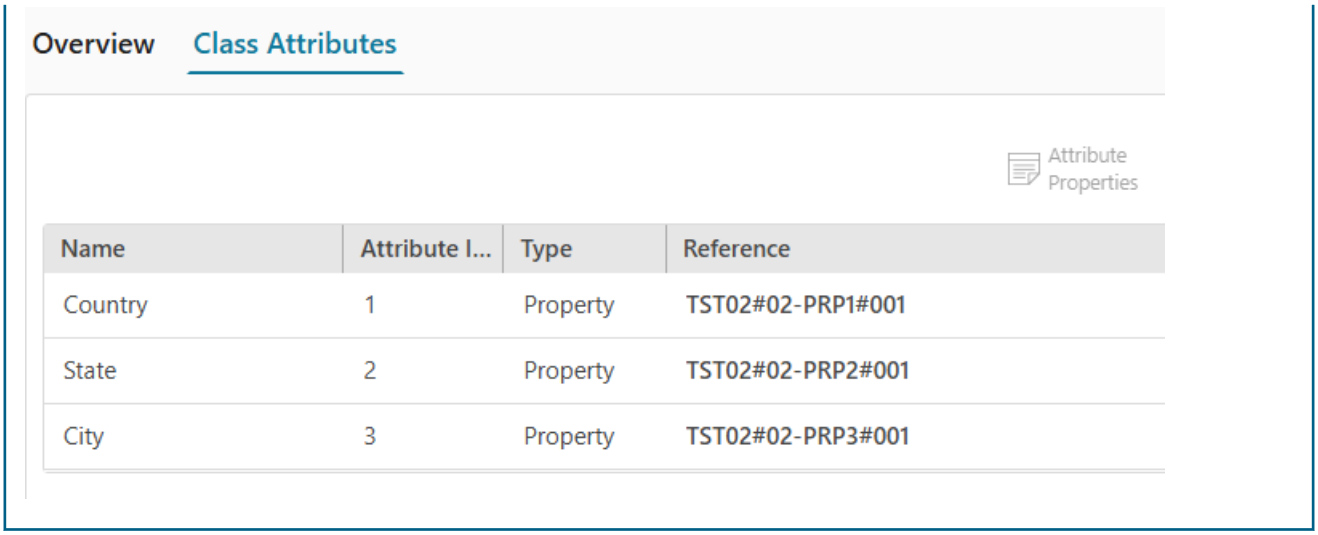
1. In Business Modeler IDE, open the **Audit Definition** editor.

2. Navigate to the classification event and select the **Is Active?** check box.

Allow auditing for class attributes

To log updates for class attributes, you must permit auditing for each class attribute using the JSON file.

> Example:
>
> For a class with IRDI TST02#01-CLS05#001, you want to allow auditing for the class attributes **Country** and **City**.



Procedure

1. Create a JSON file as follows:

```
{
"SchemaVersion": "1.5.0",
"Locale": "en_US",
"ClassDefinitions": [
    {
    "Update": ”TST02#01-CLS05#001",
    "Values": {
    "ClassAttributeOptions": [
        {
            "AttributeIRDI": "TST02#02-PRP1#001"
            "IsAuditable": true
        },
        {
            "AttributeIRDI": "TST02#02-PRP2#001"
            "IsAuditable": false
        },
        {
            "AttributeIRDI": "TST02#02-PRP3#001"
            "IsAuditable": true
        }
      ]
    }
   }
  ]
}
```

Update the JSON file as follows:

- In the **ClassDefinitions** section, specify the IRDI of the class in the **Update** section.
- In the **ClassAttributeOptions** section, update the **AttributeIRDI** attribute with the IRDI of the class attribute that you want to allow audit logging on.
- In the **ClassAttributeOptions** section, update the **IsAuditable** attribute to **true**.

2. Upload the JSON file.

View audit logs for classification events

You can view audit logs for classification events.

Procedure

1. Open a component, and click **Audit Logs**.

   The audit logs for classification events appear in the **Classification Logs** section.

Classification preferences and utilities

eclass2json.pl

Converts the various ECLASS formats to JSON or, alternatively, converts JSON files to BMEcat format. Specifically:

- Converts ontoML XML ECLASS hierarchy data to JSON format
- Converts BMEcat XML property record data to JSON format
- Converts JSON property records to BMEcat XML format

The **eclass2json** utility is a platform-independent Perl script that can be found in the following directory:

\\TR\bin\eclass2json

Syntax

eclass2json -inputFile: -objectType:
-h

Arguments

-inputFile:

Contains the source file for the conversion. This can be an ontoML XML file, a BMEcat XML file or a JSON file, depending on the conversion taking place.

-objectType:

Must be one of the following:

unitsmap

Creates an internal unit conversion file necessary for converting from ontoML files to JSON. This conversion is required once per ECLASS release.

definitions

Used to convert the following administrative objects in the order given:

1. Key-LOVs
2. Properties
3. Aspects
4. Blocks
5. Application classes
6. Classification classes

data

Used to convert BMEcat XML property records to JSON format.

json2BMEcat

Used to convert JSON format property records to BMEcat XML format.

-h

Displays assistance in using the utility.

Environment

This utility must be run in the Teamcenter shell environment. It is configured in the **eclass2json.properties** file contained in the same directory as the utility. The properties file contains directions on how to set the properties. In this property file you configure such things as:

- The status of the object being converted, for example, **Develop** or **Released**.
- Whether existing objects are skipped for conversion.
- Whether the converted object is to be added as a new release.

- The name of the top level node.
- The ECLASS level.
- The number of objects output in each file.

runClsAITraining

Runs the engine that trains classification data to create a model that is used for automatic class suggestions when classifying classes. Using artificial intelligence, classes are suggested based on the probability of them being an appropriate match.

Syntax

runClsAITraining -u=*user-name* {**-p=***password* | **-pf=***password-file*} **-g=***group-name*

-h

Arguments

-u

Specifies the user ID.

This is generally a user with administration privileges.

-p

Specifies the password.

This argument is mutually exclusive with the **-pf** argument.

-pf

Specifies the password file.

This argument is mutually exclusive with the **-p** argument.

-g

Specifies the group associated with the user.

If used without a value, the user's default group is assumed.

-h

Displays help for this utility.

The **-filename** argument that was available in earlier releases is no longer supported.

Environment

This utility must be run in the Teamcenter shell environment. It is found in the following directory:

*Teamcenter-root (TR)***\classification\ai**

cls_ai_auto_classify

Classifies many objects at the same time using AI. Objects classified in this way are sent to a workflow where administrators can check the validity of the AI predictions and make changes to the classification or reject the suggested classification.

Syntax

cls_ai_auto_classify -u=*user-name* {**-p=***password* | **-pf=***password-file*} **-g=***group-name*

-startDate -endDate -objectType -classifyObjects -outputPath -item-h]

Arguments

-u

Specifies the user ID.

This is generally a user with administration privileges.

-p

Specifies the password.

This argument is mutually exclusive with the **-pf** argument.

-pf

Specifies the password file.

This argument is mutually exclusive with the **-p** argument.

-g

Specifies the group associated with the user.

If used without a value, the user's default group is assumed.

-startDate

Restricts classification AI training data to objects created after the date specified.

-endDate

Restricts classification AI training data to objects created before the date specified.

-objectType

Restricts classification AI training data to objects with a specified object type.

-classifyObjects

Classifies objects that have a maximum probability above the percentage set in the **CLS_AI_Engine_Probability_Cutoff** preference.

Running the utility without this argument runs it in a dry run mode. A report containing the object, its classification, the probability, and an indication of whether the utility is capable of classifying the object, is stored in the specified location.

-outputPath

Specifies the file path where the auto-classify report is stored.

-item

Specifies a suffix for the training model output.

-h

Displays help for this utility.

Environment

This utility must be run in the Teamcenter shell environment.

clsutility

Performs administrative tasks when the presentation layer is installed. Using a **multitude of arguments**, you can create, update, delete, and migrate data. You can also use this utility to **display the node hierarchy** in various levels of detail.

Syntax

clsutility -u=*user-name* {**-p=***password* | **-pf=***password-file*} **-g=***group-name*

[**-create** | **-delete** | **-save** | **-find** | **-ask** | **-add** | **-import** | **-search** | **-set** | **-update** | **-get** |

-remove | **-describe** | **-migrate** | **-list** | **-h**]

Arguments

The **clsutility** utility contains many arguments and subarguments whose descriptions and syntax you can find in the help contained within the utility.

- List the utility help in the customary fashion:

```
clsutility -h
```

- List the utility help for subarguments as follows:

  `clsutility -`*`subargument`* `-h`

  For example:

  ```
  clsutility -create -h
  ```

  Running this command lists the help for all the subarguments of the **-show** command, such as the following:

```
USAGE

clsutility
  -u=<username>
  {-p=<password> | -pf=<password_file> }
  -g=<group>
  -output=<output file name>
  <command> <sub-command> <command options...>
  ------------------------------------------------------
  For the command "-create" the following sub-commands are available:
    -default_objects            [Creates default Classification Context, Hierarchy and
Scheme objects.]
    -node                       [Creates a Classification Hierarchy node object.]
    -class_definitions          [Creates a Classification standard taxonomy Class.]
    -property_definitions       [Creates a Classification standard taxonomy Property.]
    -keylov_definitions         [Creates a Classification standard taxonomy Keylov.]
    -classification_object      [Creates a new instance of Classification object.]
    -classification_objects     [Creates a set of Classification objects.]
    -node_hierarchy             [Creates a Classification node hierarchy from
Classification hierarchy.]
    -node_definitions           [Creates a Classification standard taxonomy Master
node.]
    -view_definitions           [Creates Classification Standard Taxonomy View
Definitions.]
```

Environment

This utility must be run in the Teamcenter shell environment.

clsutility reference tables

All **clsutility** commands begin with the following validation:

clsutility -u=*user-name* {**-p=***password* | **-pf=***password-file*} **-g=***group-name*

Add the following to the command to perform the required actions. Optional arguments are displayed in brackets ([ ]).

All hierarchy definitions

Use this command	To
-save -classification_definitions -request=*JSON-file-name*	Create or update key-LOV, property, class, or node definitions described in the given JSON file. This argument is available beginning with schema 1.5.0. and can be used to import all previous hierarchy definitions.

Key-LOV definitions

Use this command	To
-create -keylov_definitions -request=*JSON-file-name*	Create key-LOVs described in the given JSON file.
-delete -keylov_definitions -request=*JSON-file-name*	Delete key-LOVs described in the given JSON file.
-update -keylov_definitions -request=*JSON-file-name*	Update key-LOV definition. If it does not exist, the utility creates a new key-LOV based on the data in the JSON file.
-update -status -request=*JSON-file-name*	Update the key-LOV definition status.

Property definitions

Use this command	To
-create -property_definitions -request=*JSON-file-name*	Create properties described in the given JSON file.
-delete -property_definitions -request=*JSON-file-name*	Delete properties described in the given JSON file.
-find -property_definitions -request=*JSON-file-name*	Find properties described in the given JSON file.
-update -property_definitions -request=*JSON-file-name*	Update properties described in the given JSON file.
-update -status -request=*JSON-file-name*	Update the property definition status.

Class definitions

Use this command	To
-create -class_definitions -request=*JSON-file-name*	Create classes described in the given JSON file.
-delete -class_definitions -request=*JSON-file-name*	Delete classes described in the given JSON file.
-find -class_definitions -request=*JSON-file-name*	Find classes described in the given JSON file.
-update -class_definitions -request=*JSON-file-name*	Update classes described in the given JSON file.
-update -status -request=*JSON-file-name*	Update the class definition status.

Nodes

Use this command	To
-create -node_definitions -request=*JSON-file-name*	Create the hierarchy nodes described in the given JSON file. If a hierarchy node already exists, the utility updates the node with the values provided.
-delete -node_definitions -request=*JSON-file-name*	Delete the hierarchy nodes described in the given JSON file.
-update -node -id=*hierarchy-node-ID* **-name=***new-name-of-hierarchy-node* [**-descr=***new-description-for-given-node*]	Update a hierarchy node with the values provided. Update a hierarchy node with a new description.
-find -all_nodes	Find and display all nodes in the system.
-find -node -id=*hierarchy-node-ID-including-namespace*	Find and display a single node based on its ID.
-find -top_level_nodes	Display all top-level nodes in the system.
-get -classification_objects -id=*hierarchy-node-ID*	List the classification objects belonging to the given hierarchy node.
-ask -node_class_props -id=*hierarchy-node-ID*	List all properties belonging to a hierarchy node object.
-ask -node_class_props -id=*hierarchy-node-ID* **-name=***name-of-property-including-prefix*	List all properties belonging to a hierarchy node object with a given name.
-ask -node_parent -id=*hierarchy-node-ID* [**-full_hierarchy**]: lists all ancestor nodes of a hierarchy node	List the parent of a given hierarchy node object.
-ask -node_ancestors -id=*hierarchy-node-ID*	List all parents of a given hierarchy node object.
-ask -node_children -id=*hierarchy-node-ID* [**-type=***{***0***: group* | **1***: master* | **3***: all}*]	List all child nodes of a given type for the specified hierarchy node object and optionally, for the given type.
-ask -is_top_level_node -id=*hierarchy-node-ID*	Check whether a hierarchy node is a top-level node.
-ask -is_descendent_node -id=*hierarchy-node-ID* **-ancestor_id=***ancestor-hierarchy-node-ID*	Check whether a hierarchy node is a descendent of another node.
-ask -node_type -id=*hierarchy-node-ID*	Display the type of the given hierarchy node.
-ask -node_instances -id=*hierarchy-node-ID*	List the classification objects belonging to the given hierarchy node, including all the objects belonging to child nodes.
-ask -node_class_props -id=*hierarchy-node-ID*	List properties belonging to a hierarchy node.
-ask -node_class_prop -id=*hierarchy-node-ID* **-name=***name-of-property-being-sought-including-prefix*	List all properties belonging to a hierarchy node.

Use this command	To
-ask -node_attachments -id=*hierarchy-node-ID*	List all the attachments belonging to the given hierarchy node.
-ask -node_characteristic -id=*hierarchy-node-ID* **-characteristic={0**: *is-an-assembly* | **1**: *has-multiple-instances* | **3**: *is-a-leaf-node}*	List whether a given node is an assembly, has multiple instances, or is a leaf node.
-import -image -id=*hierarchy-node-ID* **-os_path=***path-to-image-file* [**-new_file_name=***new-file-name-used-by-Active-Workspace*] [**-do_update**]: updates existing primary image [**-is_primary_image**]: sets the image to primary image	Import an image for the given hierarchy node. Images are displayed in the right pane of the user interface. Teamcenter supports the following file types for class images: GIF, JPG, JPEG, PNG, BMP, SVG, and PDF.
-import -icon -id=*hierarchy-node-ID* **-os_path=***path-to-image-file* [**-new_file_name=***new-file-name-used-by-Active-Workspace*] [**-do_update**]: updates existing icon	Import an icon for the given hierarchy node. Icons are displayed in the classification hierarchy and help to visualize the class names. Teamcenter supports the following file types for node icons: GIF, JPG, JPEG, PNG, BMP, SVG, and PDF.
-remove -image -id=*hierarchy-node-ID* [**-delete_dataset**]: removes image dataset in addition to image	Remove an image attached to a node.
-remove -icon -id=*hierarchy-node-ID* [**-delete_dataset**]: removes icon dataset in addition to icon	Remove an icon attached to a node.
-get -image -id=*hierarchy-node-ID*	Identify the primary image used by the given hierarchy node.
-get -images -id=*hierarchy-node-ID*	List all the images used by the given hierarchy node.
-get -icon -id=*hierarchy-node-ID*	Identify the icon used by the given hierarchy node.

Classification objects (ICOs)

Use this command	To
-create -classification_objects -node_id=*hierarchy-node-ID* [**-obj_count=***number-of-classification-objects*] [**-prefixid=***prefix-assigned-to-new-objects*] | [**-request=***JSON-file-name*]	Create multiple classification instances.
-find -classification_objects -request=*JSON-file-name*	Export the classification properties of the classification objects listed in the JSON request

Use this command	To
	file. In the request file, the ICO object ID must be listed with its revision, for example, 123456/A.

Classification views

Use this command	To
-create -base_view_definitions	Create base views for every released application class. If a base already exists for a class, the class is skipped.
-create -base_view_definitions -class_definition *IRDI-of-class*	Create a base view for the given application class.
-create -base_view_definitions -force	Create or recreates a base view for every released application class.
-create -view_definitions -request=*JSON-file-name*	Create a user, group, or role view for a class, or modifies the order of or suppresses properties in a base view for a class.
-get -class_descriptor -request=*JSON-file-name*	List the class descriptor for specified classes.

Examine the hierarchy using the clsutility -list -hierarchy utility

For most general use cases, classification admin objects are listed in the **Classification Manager**. Additionally, you can use the **clsutility**, to list classification standard taxonomy hierarchy objects to help you better understand the classification data in the database. This method of viewing the hierarchy is based on the *class descriptor*. Whereas a class definition describes only the class, or a key-LOV definition only one key-LOV, the class descriptor compiles all the definitions required to describe a classification object (ICO). Each node of the hierarchy shares a class descriptor. The class descriptor provides a view on the data from the classification consumer perspective.

> Note:
>
> When you list all the nodes in the database, the utility lists hierarchies that are not migrated to CST, but it cannot display ICOs belonging to these hierarchies.

You can direct the output to a text file that you can open in a text editor by using the **-output=***output-file-name.txt*.

The following arguments are available when using **clsutility -list -hierarchy**:

Argument	Description
-nodeId=	Specifies the unique ID of the node for which you want to display the hierarchy. The utility displays all the parents of the given node, as well as the node itself. If a node ID is not provided, all top level nodes are displayed.

Argument	Description
	To view the children of the given node, additionally use the **-recursive** argument. If you use the **-nodeId** argument, you cannot use the **-classId** and **-classNamespace** arguments.
-classId=	Specifies the class ID of the class definition referenced from the given node. If you specify a class ID here, you must also specify a namespace in the **-classNamespace** argument. If you use this argument, you cannot use the **-nodeId** argument.
-classNamespace=	Specifies the namespace of the class definition referenced from the node. If you specify a namespace here, you must also specify a class ID in the **-classId** argument. If you use this argument, you cannot use the **-nodeId** argument.
-recursive	Displays the entire hierarchy below the specified node. You cannot use this argument with the **-showClass** argument.
-showClass	Shows detailed information about the referenced classes of the given node. You cannot use this argument with the **-recursive** argument.
-showJSON	Displays the JSON response from **CST_class_describe**, which is used to build the report of a class.
-showType	Displays the class definition of the dynamic runtime type. This argument is used for Siemens Digital Industries Software internal troubleshooting purposes.
-showICOs	Displays the ICOs of the nodes.
-showICOProperties	Displays all ICO properties if you have also specified **-showICOs**.
-icoLimit=	Limits the number of ICOs of a single node that are to be displayed.
-icoId=	Shows only the ICO with this ID.

- To view all the top nodes in the hierarchy, type:

  clsutility -list -hierarchy

- To view all the nodes in the hierarchy, type:

  clsutility -list -hierarchy -recursive

- To view all the nodes underneath a specified node, type:

  clsutility -list -hierarchy -nodeId=*node ID* **-recursive**

- To view the details of a specific class, type:

clsutility -list -hierarchy -nodeId=*node ID* **-showClass**

For example, for an ECLASS class called **AC-DC supply**, the **clsutility -list -hierarchy -recursive** command can be used to find the node ID (in this example, **0173-1#AFX043**):

```
  "27-04-06-90" [0173-1#BAB258 ==> 0173-1#01-ADP432#006] "[27-04-06-90] No-break power supply (complete, unspecified)"
}
"27-04-07-00" [0173-1#AFR649] "[27-04-07] Power supply device"{
  "27-04-07-01" [0173-1#AFX040 ==> 0173-1#01-AFR739#002] "[27-04-07-01] Continuous current supply"
  "27-04-07-02" [0173-1#AFX041 ==> 0173-1#01-AFR741#002] "[27-04-07-02] Voltage stabilizer"
  "27-04-07-03" [0173-1#AFX042 ==> 0173-1#01-AFR743#002] "[27-04-07-03] Alternating current supply"
  "27-04-07-04" [0173-1#AFX043 ==> 0173-1#01-AFR745#002] "[27-04-07-04] AC-DC supply"
  "27-04-07-05" [0173-1#AFX039 ==> 0173-1#01-AFR737#002] "[27-04-07-05] Stand-by unit"
  "27-04-07-90" [0173-1#AFX044 ==> 0173-1#01-AFR747#002] "[27-04-07-90] Power supply device (unspecified)"
}
"27-04-91-00" [0173-1#AAB638] "[27-04-91] Power supply (parts)"{
  "27-04-91-01" [0173-1#AFQ933 ==> 0173-1#01-AFR005#003] "[27-04-91-01] No-break power supply (complete, parts)"
  "27-04-91-90" [0173-1#AFZ646 ==> 0173-1#01-ADP554#006] "[27-04-91-90] Power supply (parts, unspecified)"
}
"27-04-92-00" [0173-1#AKP085] "[27-04-92] Power supply (accessories)"{
  "27-04-92-01" [0173-1#AKN587 ==> 0173-1#01-ADO360#006] "[27-04-92-01] UPS system (accessories)"
  "27-04-92-90" [0173-1#AEI674 ==> 0173-1#01-AEX753#004] "[27-04-92-90] Power supply (accessories, unspecified)"
}
```

You can then use the node ID as input for the **clsutility -list -hierarchy -nodeId=0173-1#AFX043 -showClass** command. The output resembles the following:

```
"00-00-00-00" [0173-1#4711A1] "eCl@ss ADVANCED 4711-A1"
"27-00-00-00" [0173-1#AAB572] "[27] Electric engineering, automation, process control engineering"
"27-04-00-00" [0173-1#AAB631] "[27-04] Power supply devices"
"27-04-07-00" [0173-1#AFR649] "[27-04-07] Power supply device"
"27-04-07-04" [0173-1#AFX043 ==> 0173-1#01-AFR745#002] "[27-04-07-04] AC-DC supply"
              { Class[0173-1#01-AFR745#002] "AC-DC supply"
                Status: "Released"
                01:   [0173-1#01-ADN464#006] Aspect  "Add on Documentation"
                      { Class[0173-1#01-ADN464#006] "Add on Documentation"
                        Status: "Released"
                        01:01 [0173-1#02-AAN469#002] Integer  "number of documentation"
                              IsCardinalityController
                        02:   [0173-1#02-AAQ680#006] Reference  "Additional Information"
                              CardinalityController: 0173-1#02-AAN469#002
                              { Class[0173-1#01-ADN356#006] "Additional Information"
                                Status: "Released"
                                01:01 [0173-1#02-AAC312#001] Date  "calendrical validity from"
                                02:   [0173-1#02-AAN466#001] L10NString  "Description"
                                03:01 [0173-1#02-AAN467#003] String  "Country/region classification the document"
                                      { KeyLOV[0173-1#09-AAD528#002] String (entries:246)
                                        001: "AF"  "Afghanistan                    "  "0173-1#07-AAS015#001" ""
                                        002: "AX"  "Åland                         "  "0173-1#07-AAS156#001" ""
                                        003: "AL"  "Albania                        "  "0173-1#07-AAS157#001" ""
                                        004: "DZ"  "Algeria                        "  "0173-1#07-AAS158#001" ""
                                        005: "AS"  "American Samoa                 "  "0173-1#07-AAS159#001" ""
                                        006: "AD"  "Andorra                        "  "0173-1#07-AAS161#001" ""
                                        007: "AO"  "Angola                         "  "0173-1#07-AAS162#001" ""
                                        008: "AI"  "Anguilla                       "  "0173-1#07-AAS163#001" ""
```

This output displays all the parent nodes of **0173-1#AFX043**, as well as all the classes (application, block, aspects) referenced by it, the properties, and the key-LOVs included in the class. The properties are listed as follows:

```
15:04 [0173-1#02-AAB774#006] Double ("V")  "max. 1. output voltage with AC"
```

1 2 3 4 5

1. Index:internal code
2. IRDI
3. Data type
4. Unit
5. Property name

If key-LOVs are used repeatedly in a class, their details are listed the first time they are used.

```
04:02 [0173-1#02-AAN350#003] String  "Part relation to the main device"
    { KeyLOV[0173-1#09-AAD520#003] String (entries:7)
      1: "accessory                "  "accessory                "  "0173-1#07-AAR847#001" ""
      2: "counterpart              "  "counterpart              "  "0173-1#07-AAR849#001" ""
      3: "Label parts (inscription)"  "Label parts (inscription)"  "0173-1#07-AAR848#002" ""
      4: "Wartungsteil             "  "maintenance part         "  "0173-1#07-AAR846#001" ""
      5: "resolve part             "  "resolve part             "  "0173-1#07-AAR844#001" ""
      6: "spare                    "  "spare                    "  "0173-1#07-AAR845#002" ""
      7: "system component         "  "system component         "  "0173-1#07-AAR843#002" ""
    }
```

All subsequent uses of the key-LOV are abbreviated with a reference to the first usage:

```
04:02 [0173-1#02-AAN350#003] String  "Part relation to the main device"
      KeyLOV[0173-1#09-AAD520#003] { ADDITIONAL USE: For details see output above }
```

- To find the node from which you want to begin displaying the hierarchy, assuming you know the class ID and the class namespace, type:

  clsutility -list -hierarchy -classId= *class ID* **-classNamespace=***class namespace*

  If the class ID of the **AC-DC supply** class is **AFR745** and the namespace is **0173-1**, type:

  clsutility -list -hierarchy -classId=AFR745 -classNamespace=0173-1

  The output is as follows:

```
"00-00-00-00" [0173-1#4711A1] "eCl@ss ADVANCED 4711-A1"
"27-00-00-00" [0173-1#AAB572] "[27] Electric engineering, automation, process control engineering"
"27-04-00-00" [0173-1#AAB631] "[27-04] Power supply devices"
"27-04-07-00" [0173-1#AFR649] "[27-04-07] Power supply device"
"27-04-07-04" [0173-1#AFX043 ==> 0173-1#01-AFR745#002] "[27-04-07-04] AC-DC supply"
```

- To display the JSON file from which the output information is drawn, type:

  clsutility -list -hierarchy -showClass -showJSON

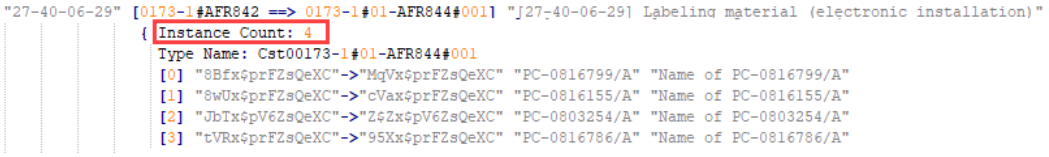
This argument functions only with the **-showClass** argument.

The output includes the description of the class as well as the JSON that the user interface uses to display the class.

- To display all the ICOs of a specific node, type:

  clsutility -list -hierarchy -nodeId= 0173-1#AFR842 -showICOs

  All the ICOs are displayed in addition to the complete hierarchy, with the number of instances displayed for each class:



  To display a specific ICO from the preceding list, type

  clsutility -list -hierarchy -showICOs -icoId=*ICO ID*

  This is particularly useful when combined with **-showICOProperties**.

- To display no more than *x* number of ICOs within a class, type

  clsutility -list -hierarchy -showICOs -recursive -icoLimit=*x*

View output in a text editor

The text files created by this utility can be extremely large. To assist you in manipulating these files, you can try enabling the collapsing of code in a text editor. This can be done, for example, by viewing the text file as C++ code, or a similar viewing feature depending on the text editor. This adds the ability to collapse at every curly bracket in the output of the hierarchy. Collapsing unwanted portions of the output makes it easier to find what you are looking for.

Setting classification preferences

The following preferences are available to configure your classification environment.

ICS_classifiable_types

Stores the business object types that can be classified. If your company works with custom business objects, you must add the item type to this preference to be able to classify it. If you want to classify, for example, items only, you must remove the **Item Revision** entry from the list of classifiable types.

AWC_classification_facets_threshold

Sets the number of search facets displayed using the preference. The default value is 200. The facets are sorted with the most common facets at the top. Exercise caution when setting the number of returned facets as a very large number can cause a decrease in performance.

CLS_is_presentation_hierarchy_active

Specifies whether traditional classes or the presentation hierarchy (nodes) are visible in the classification hierarchy. If this preferences is set to **true**, the presentation node hierarchy is visible. If it is set to **false**, traditional classes are visible. The default value of the preference is **true**.

This user preference is not delivered by default and must be added manually after installing the presentation layer (**Next Generation Classification Server** installation option).

CLS_auto_sync_node_hierarchy

Allows the system to mirror operations performed on the classification storage class hierarchy onto the node hierarchy in the presentation layer.

The default value of the preference is **true**.

CLS_search_similar_wso_props_enabled

Specifies the list of workspace object properties that are displayed in the **Filters** pane when using the **Search Similar** feature.

AWC_cls_object_filter_sorting

Defines how the classification hierarchy is sorted in the **Filter** panel. Valid values are:

- **Ascending**

  Sorts the hierarchy in alphabetical order, from A to Z.

- **Descending**

  Sorts the hierarchy in reverse alphabetical order, from Z to A.

- **Count** (default)

  Sorts the hierarchy based on the number of instances for the given filter value.

CST_supported_eclass_releases

The following preferences are required when migrating from traditional basic classes to advanced (classification standard taxonomy, CST) classes using the **clsutility -migrate -classification2cst** utility:

CST_default_namespace

Specifies the namespace of all CST objects created by the migration utility.

CST_default_migration_status

Specifies the status of the CST objects created by the migration utility. Valid values are **Develop**, **Released**, or **Retired**. Objects set to **Released** can no longer be modified.

The following preferences are used to configure classification artificial intelligence (AI):

CLS_AI_Enable_AI_Engine

Set this preference to **true** to enable class suggestions.

TC_Microservice_Base_URL

Specifies the URL of the microservice that supports classification AI. This is the same path that you specify during installation of classification AI.

CLS_AI_Object_Properties

Specifies the properties that are extracted from the Teamcenter database and used to train the AI engine. These properties form the basis for the class suggestions offered when classifying objects.

By default this property is set to:

object_name

object_desc

object_type

owning_user

owning_group

CLS_AI_Enable_Geolus

Set this preference to **true** to enable the additional comparison of shape parameters using the Geolus search engine with classification AI.

CLS_AI_Geolus_Max_Num_Results

Limits the number of results returned by the Geolus search engine. By default, this preference is set to **1000**.

CLS_AI_Engine_Probability_Cutoff

Specifies the percentage probability above which classes are considered appropriate suggestions for classification AI. If too many class suggestions are displayed or you would like to increase the accuracy of the classes displayed, you can increase the value of this preference. By default, this preference is set to **10**.

CLS_AI_Engine_Suggestions_Cutoff

Specifies the maximum number of suggestions displayed to the user. The default value is **10**. This preference is not delivered with the software. You must add it manually.

CLS_AI_Query_Start_Date

Specifies the earliest modification date of the data to include in training. Only objects created or modified after this date are included in the training. If this date is not set, there is no restriction on objects used in training.

CLS_AI_Query_End_Date

Specifies the latest modification date of the data to include in training. Only objects created or modified before this date are included in the training. If this date is not set, there is no restriction on objects used in training.

CLS_AI_Query_Object_Type

Specifies the type of data to include in training. Only objects of the given type (and any child objects of this type) are included for training. If left blank, the value of this preference is **WorkspaceObject** by default.