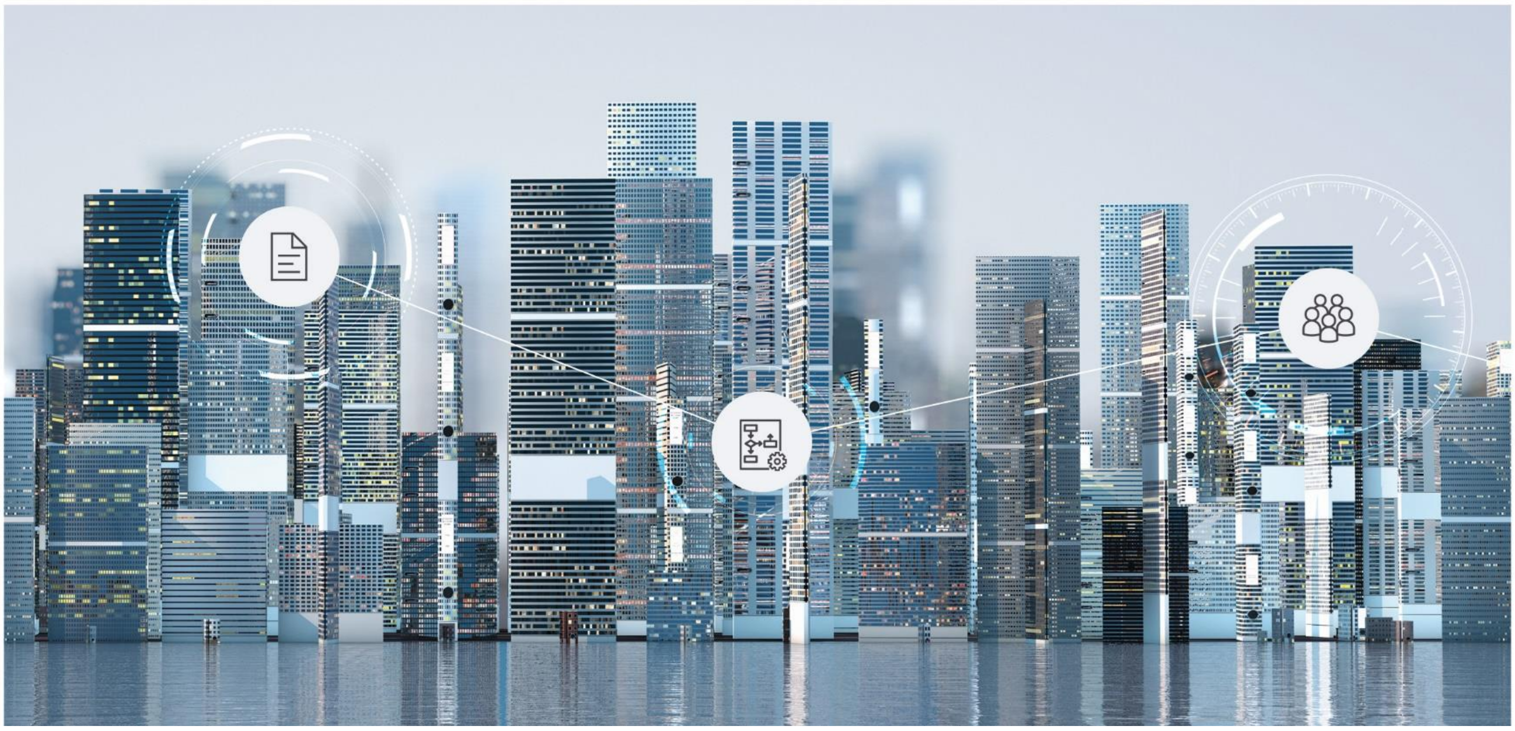


White Paper

Fabasoft app.ducx Extensions for Folio Developers

2022 June Release

Contents

1 Introduction 6

2 Working With Fabasoft app.ducx Projects Using Eclipse 6

2.1 Adding Additional Contents 6

2.2 Fabasoft app.ducx Builders 6

2.3 Exporting a Fabasoft app.ducx Project 7

2.4 Environment Variables 8

2.5 Creating Your Own Setup Kit 9

3 app.ducx Object Model Language 9

3.1 Defining the Base Class 9

3.2 Updating an Existing Component Object Instance 9

3.3 Defining Proprietary Types and Property Classes 10

3.4 Software Products, Software Editions and Software Solutions 11

3.4.1 Software Product 11

3.4.2 Software Edition 12

3.4.3 Software Solution 12

3.4.4 Example 12

4 app.ducx User Interface Language 13

4.1 Defining Portals 13

5 app.ducx Use Case Language 14

5.1 Virtual Application Context Actions 14

5.2 Extending a Virtual Application 14

5.3 Overriding an Existing Use Case Implementation 15

5.4 Use Case Wrappers 15

5.5 Use Case Wrappers With Data 16

5.6 Use Case Wrappers (Old Style) 17

6 app.ducx Organizational Structure Language 18

6.1 Defining a Position 18

6.2 Defining an Organizational Unit 19

6.3 Extending an Organizational Unit 19

6.4 Defining an Access Type 20

7 app.ducx Customization Language 20

7.1 Using “add” and “override” of a Software Solution or Edition (Deprecated) 20

8 Testing and Debugging 21
8.1 Tracing in Fabasoft app.ducx Projects 21
9 Sample: Creating a Wizard 21
10 Transformation of Legacy Software Components 23
10.1 Creating an Fabasoft app.ducx Project From an Existing Software Component 23
10.2 Object Model Language 24
10.3 Resource Language 25
10.4 User Interface Language 25
10.5 Use Case Language 26
10.6 Organization Structure Language 26
10.7 Business Process language 26
10.8 Customization Language 26
10.9 Expression Language 26
11 Profiles 27
11.1 Cloud App Profile Restrictions 27
11.1.1 Object Model Language 27
11.1.2 User Interface Language 28
11.1.3 Use Case Language 29
11.1.4 Organization Structure Language 29
11.1.5 Business Process language 29
11.1.6 Customization Language 30
11.2 Base App Profile Restrictions 30
11.2.1 Object Model Language 30
11.2.2 User Interface Language 31
11.2.3 Use Case Language 32
11.2.4 Organization Structure Language 32
11.2.5 Business Process language 32
11.2.6 Customization Language 33
11.3 Enterprise App Profile Restrictions 33
11.3.1 Object Model Language 33
11.3.2 User Interface Language 34
11.3.3 Use Case Language 35
11.3.4 Business Process language 35
11.3.5 Customization Language 35

1 Introduction

Fabasoft app.ducx is Fabasoft's powerful use case-oriented application development system for developing composite content applications (CCAs).

The Fabasoft app.ducx reference documentation series consists of the following three consecutive documents:

- Fabasoft app.ducx for Cloud Developers
  This document explains all concepts and use cases that are relevant for Cloud and Folio developers. This document is aimed at software developers interested in exploring the tremendous potential of Fabasoft app.ducx.
- **Fabasoft app.ducx Extensions for Folio Developers**
  This document describes extensions that are relevant for Folio developers. It is highly recommended to read "Fabasoft app.ducx for Cloud Developers" beforehand, because all the presented concepts are also directed at Folio developers.
  Note: You are currently reading this document.
- Fabasoft app.ducx Extensions for Java Developers
  This document provides an overview of using Java in your app.ducx project. Java can only be used along with Folio. It is highly recommended to read "Fabasoft app.ducx Extensions for Folio Developers" beforehand.

2 Working With Fabasoft app.ducx Projects Using Eclipse

If you are a Folio developer, you have to consider following additional aspects when working with app.ducx projects.

2.1 Adding Additional Contents

Additional contents may be specified within Fabasoft app.ducx project settings. It is solely for additional contents, e.g. help contents (Compiled Help Files) or contents for a specific platform, e.g. Windows x64 only. These contents will be part of your software component.

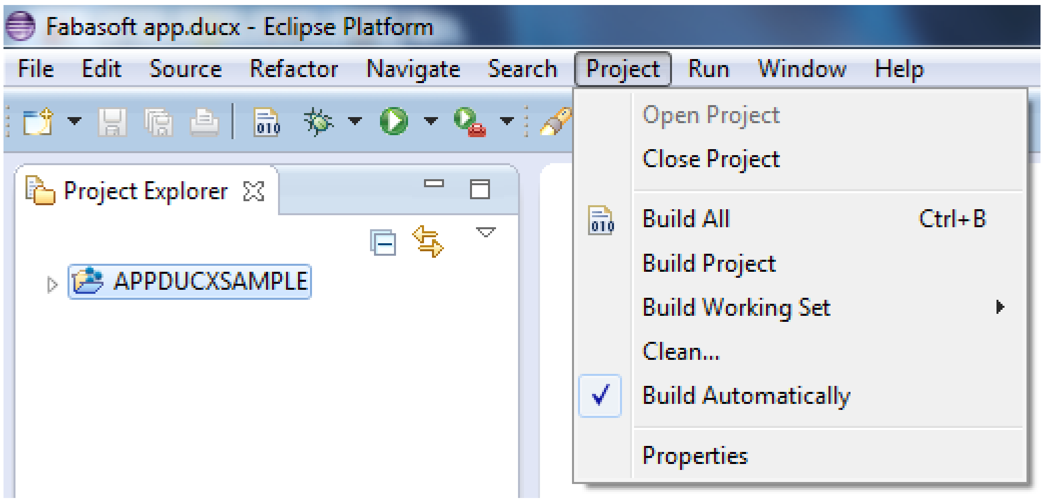
Contents are simply added by dropping a bunch of files from the file system. Additional properties may be set after drop. The basename is initially constructed from the basename of the file. The type is `CCT_NORMAL` by default. The basename may be changed.

Fabasoft app.ducx tries to identify available variables. The longest match is used. Variables include environment variables and variables available within in Eclipse.

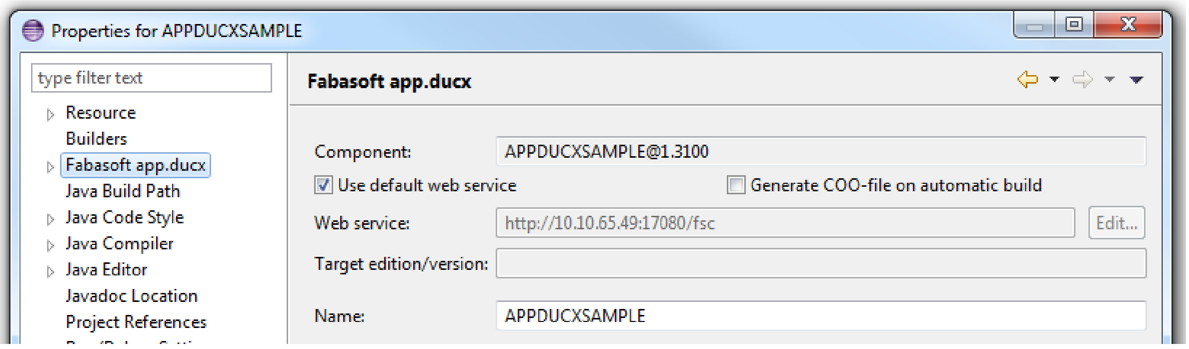
Note: When using a build process out of eclipse, Eclipse variables might not be available.

2.2 Fabasoft app.ducx Builders

Running app.ducx in Eclipse requires the setting "Project" > "Build Automatically" to be enabled, so that cross-referencing between different source code sections can be established. This setting will be automatically enabled once after installing the Fabasoft app.ducx feature.



Generating output of a Fabasoft app.ducx project is done by the Fabasoft app.ducx builder. This builder can be configured to run on each automatically triggered build by the Eclipse Platform. The corresponding setting ("Generate COO-file on automatic build") has to be set in each project's preferences. By default, this setting is disabled because generating output is not necessary on each save operation in Eclipse.



Component: APPDUCXSAMPLE@1.3100

[✓] Use default web service

[ ] Generate COO-file on automatic build

Web service: http://10.10.65.49:17080/fsc

Target edition/version:

Name: APPDUCXSAMPLE

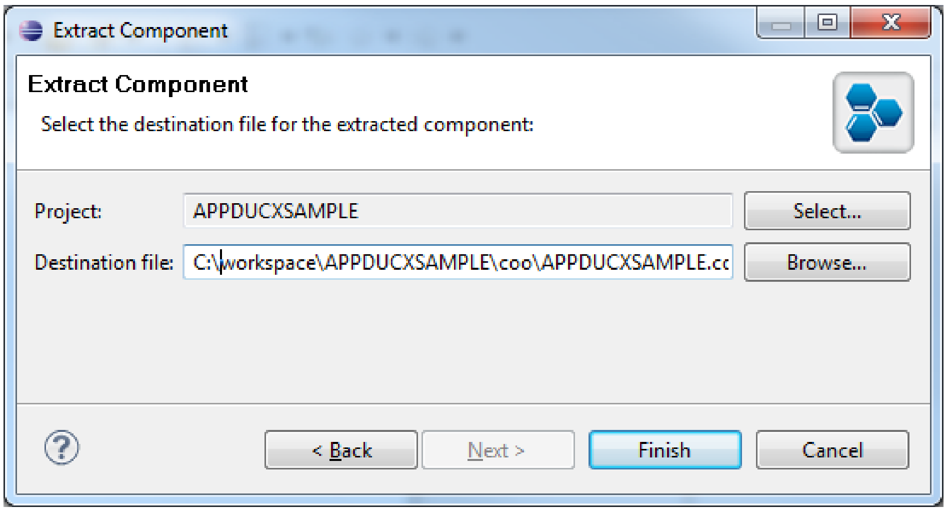
2.3 Exporting a Fabasoft app.ducx Project

Exporting a Fabasoft app.ducx project allows you to distribute your project to customers. A Fabasoft app.ducx project represents a single software component. Related software components can be encapsulated in a software product. Several software products build up a software edition. A software edition is a complete product. To customize a software edition depending on customer wishes is done with a software solution.

Software products, software editions and software solutions are defined as instances within a Fabasoft app.ducx project.

Exporting Software Components

In Eclipse, open the "File" menu and click "Export". Navigate to "Fabasoft app.ducx" > "Extract Component" to export your Fabasoft app.ducx project to a container file (with a `.coo` extension) that can be installed in another domain by loading it using the management tool.


Extract Component

Extract Component

Select the destination file for the extracted component:

Project: APPDUCXSAMPLE

Destination file: C:\workspace\APPDUCXSAMPLE\coo\APPDUCXSAMPLE.cc

Exporting Software Products, Software Editions, and Software Solutions

In Eclipse, open the “File” menu and click “Export”. Navigate to “Fabasoft app.ducx” > “Extract Solution” to export a software product, software edition or software solution of your Fabasoft app.ducx project.

2.4 Environment Variables

The following Java-specific environment variables may be used in the Fabasoft app.ducx project file. If this is the case, make sure that the environment variables are also set in the build environment or that the corresponding values are defined in the Apache Ant `build.xml`.

Environment variable for the java compiler.

- To ensure correct java libraries, the java compiler has to be specified by the environment variable `DUCX_JAVA_HOME[_X]` This variable is used to specify the java compiler required for specific versions of Fabasoft Folio. X stands for the java version. Currently most versions require Java 8 so to be sure that every java compiler from 8 up to the current version generates compatible code, set the variable in an appropriate way (e.g. `DUCX_JAVA_HOME_8="C:/Java/jdk1.8.0_332"`).
- The order of variable checking is:
  - *`DUCX_JAVA_HOME_X`*
  - *`DUCX_JAVA_HOME`*
  - *`JAVA_HOME`*

If you compile a Fabasoft app.ducx project containing Java source code, the manifest of the generated JAR file can be populated with following environment variables:

- `DUCX_SPEC_TITLE=SPEC_TITLE` (string)
  Defines the title of the specification.
- `DUCX_SPEC_VENDOR=SPEC_VENDOR` (string)
  Defines the organization that maintains the specification.
- `DUCX_SPEC_VERSION=SPEC_VERSION` (string)
  Defines the version of the specification.
- `DUCX_IMPL_TITLE` (string)
  Defines the title of the implementation.

- `DUCX_IMPL_URL` (string)
  Defines the URL from which the implementation can be downloaded from.
- `DUCX_IMPL_VERSION` (string)
  Defines the version of the implementation.
- `DUCX_IMPL_VENDOR` (string)
  Defines the organization that maintains the implementation.
- `DUCX_IMPL_SEALED` (`true`|`false`)
  Should be set `true` because all classes defined in all package are archived in the same Fabasoft app.ducx JAR file.

2.5 Creating Your Own Setup Kit

To create a setup kit including your own software solution perform the following steps:

1. Copy the Fabasoft Folio kit on a writeable media.
2. Copy your software solution in the `Setup` directory.

The base product extended by your own software solution can now be installed or updated using `setup.exe` or `setup.sh`.

Note:

- The property `COOSYSTEM@1.1:solallowedbaseeditions` of a software solution contains the references of allowed base software editions for this software solution. At least one of the allowed base software editions must be installed in the domain to be able to install this software solution. Leave this property empty, if the software solution should be a stand-alone solution instead of an add-on solution. Keep in mind that only one stand-alone software edition or software solution can be installed in a domain.
- Make sure that all COO files of your additionally needed software components (Fabasoft app.ducx projects) are copied in the software solution folder. The software components have to be referenced in the property `COOSYSTEM@1.1:prodcomponents`.

3 app.ducx Object Model Language

This chapter outlines object model language features that are relevant for Folio developers.

3.1 Defining the Base Class

There is no restriction for defining the base class.

3.2 Updating an Existing Component Object Instance

Syntax

```
update instance <reference> {
  ...
}
```

Using the `update instance` keywords, it is possible to update selected properties of an existing component object.

The `update instance` keywords must be followed by the reference or the object address of the component object to be updated. You can use generic assignment statements inside the `update instance` block to assign values to properties of the instance. The values of the referenced properties are replaced by the values inside the block.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  // Exclude exclusively order and invoice objects from the list of objects allowed
  // on the user's desk
  update instance objchildren {
    attrnotallowed<attrallclass, attrallcomponent> = {
      { Order, APPDUCXSAMPLE@200.200 },
      { Invoice, APPDUCXSAMPLE@200.200 }
    }
  }
}
```

3.3 Defining Proprietary Types and Property Classes

In some rare cases it is necessary to define special type instances. This is done by creating an instance of a type class. Properties using this type can be declared by using a special syntax, specifying the property class and the new type.

It is even possible to declare a special property class to be able to specifiy new property attributes. This class has to be a subclass of an existing property class.

The software component `NUMERATOR@1.1001` uses this method to define special property types for key numerators. See the sample code how to use these.

Note: Using proprietary types and property definition classes might result in unexpected behavior of the generic UI controls, so the general advice is not to use this feature unless you are really familiar with the existing object model.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import NUMERATOR@1.1001;
  /*
   * new type to be used in this project
   */
  instance TypeStringDef ZIPCODE {
  }
  /*
   * property definition class for properties of new type
   */
  class AttributeZipcodeDef : AttributeStringDef {
    programmatic = true;
    attrinitobjs<attrobjmember, attrinitobjval> = {
      { attrtype, ZIPCODE }
    }
    boolean digitsonly;
  }
  /*
   * class using the new property and type definitions
   */
  class Location {
    AttributeStringDef<ZIPCODE> anyzipcode;
    AttributeZipcodeDef<ZIPCODE> germanzipcode {
      digitsonly = true;
    }
    KeyNumerator<INTEGER> locnumber {
      KeyEntryList<NUMERATOR@1.1001:objclass, keyattrlist> = {
        { Location, objowner }
      }
    }
  }
}
```

3.4 Software Products, Software Editions and Software Solutions

Software products, software editions and software solutions are defined as instances within a Fabasoft app.ducx project. To export a software edition or software solution open the “File” menu, click “Export” and click “Fabasoft app.ducx” > “Extract Solution”. Click “Next”. Select a project that contains a software edition or software solution, specify a folder and click “Finish”. All files of the software edition or software solution are exported as a single ZIP file. Alternatively, an Apache Ant task can be used.

Note: Software products and software editions are for Fabasoft internal use only. Software solutions are designed for packaging software projects of Fabasoft partners and customers.

Available properties can be found in the following chapters.

3.4.1 Software Product

- `COOSYSTEM@1.1:prodname`
  Defines the name of the software product.
- `COOSYSTEM@1.1:prodverscode`
  Defines the version of the software product.
- `COOSYSTEM@1.1:prodstate`
  Defines whether the software product is in development (`PRODST_DEVELOP`) or is in production (`PRODST_INSTALLED`).
- `COOSYSTEM@1.1:prodcopyright`
  Defines the copyright of the software product.
- `COOSYSTEM@1.1:prodcopyrightbmp`
  Defines the copyright image of the software product.
- `COOSYSTEM@1.1:prodcomponents`
  Defines references of software components that are contained in the software product.
- `COOSYSTEM@1.1:proddemocomponents`
  Defines references of demo software components that are contained in the software product.
- `COOSYSTEM@1.1:produnselcomponents`
  Defines references of software components that are not installed by default when installing the software product.
- `COOSYSTEM@1.1:prodexcomponents`
  Defines references of software components that are not contained any longer in the software product.
- `COOSYSTEM@1.1:prodadminfiles`
  Contains administrative COO files.
- `COOSYSTEM@1.1:prodconfigexpr`
  Contains app.ducx Expression Language to customize the Fabasoft Folio Domain.

- `COOSYSTEM@1.1:compcontents`
  Contains content files like help files.

3.4.2 Software Edition

- `COOSYSTEM@1.1:prodbaseprodobjs`
  Contains the software products, the software edition is based on.
- `COOSYSTEM@1.1:editallowedbaseeditions`
  Contains the references of allowed base software editions for this software edition. At least one of the allowed base software editions must be installed in the Fabasoft Folio Domain to be able to install this software edition. Leave this property empty, if the software edition should be a stand-alone edition instead of an add-on edition. Fabasoft Folio Compliance is a typical stand-alone edition and Fabasoft app.ducx is a typical add-on edition. Keep in mind that only one stand-alone software edition or software solution can be installed in a Fabasoft Folio Domain.
- `COOSYSTEM@1.1:proddemodata`
  Contains the references of demo data objects.

Note: Additionally, properties of the software product are also available in a software edition.

3.4.3 Software Solution

- `COOSYSTEM@1.1:prodbaseprodobjs`
  Contains the software products the software solution is based on.
- `COOSYSTEM@1.1:solallowedbaseeditions`
  Contains the references of allowed base software editions for this software solution. At least one of the allowed base software editions must be installed in the Fabasoft Folio Domain to be able to install this software solution. Leave this property empty, if the software solution should be a stand-alone solution instead of an add-on solution. Keep in mind that only one stand-alone software edition or software solution can be installed in a Fabasoft Folio Domain.
- `COOSYSTEM@1.1:proddemodata`
  Contains the references of demo data objects.

Note: Additionally, properties of the software product are also available in a software solution.

3.4.4 Example

A solution may look like the following example.

Example

```
instance SoftwareSolution SolutionSample {
  prodname = "Sample Solution";
  prodverscode = 1005;
  prodstate = PRODST_INSTALLED;
  prodcopyright = file("resources/copyright.txt");
  prodcopyrightbmp = file("resources/copyright.bmp");
  prodadminfiles<ncname, nccont> = {
    { "baseaclcfg.coo",  file("resources/config/baseaclcfg.coo") },
    { "baseguicfg.coo",  file("resources/config/baseguicfg.coo") },
    { "baseseccfg.coo",  file("resources/config/baseseccfg.coo") }
  }
  proddemodata = "APPDUCXSAMPLE@200.200:DemoDataForSampleSolution";
}
instance DemoData DemoDataForSampleSolution {
  ddcustomizingexprs<ddename, ddeexpr> = {
    { "FolioDataExpression.exp",
      file("resources/imports/FolioDataExpression.exp")
    }
  }
  ddimports<ddidatasource, component> = {
    {
      file("resources/imports/FOLIOIMPORT_00_Group.csv"),
      APPDUCXSAMPLE@200.200
    }
  }
}
```

4 app.ducx User Interface Language

This chapter outlines user interface language features that are relevant for Folio developers.

4.1 Defining Portals

Syntax

```
portal reference {
  pane reference {
    colorscheme = colorscheme;
    page reference {
      application = application;
    }
  }
  ...
}
```

The app.ducx user interface language allows you to define custom portals in your software component.

In Fabasoft Folio, a portal is comprised of three distinct elements:

- the *Portal* (`FSCVPORT@1.1001:Portal`) object itself
- one or more *Portal Panes* (`FSCVPORT@1.1001:Pane`) that are part of the portal
- one or more *Portal Pages* (`FSCVPORT@1.1001:Page`) that are part of a portal pane

The keyword `portal` is used to define a portal. It must be followed by a reference and curly braces. The portal may contain multiple portal panes, which are defined inside the `portal` block using the `pane` keyword.

Each `pane` block must have a reference followed by curly braces. Inside the `pane` block, you may use the `colorscheme` keyword to assign a predefined color scheme to the portal pane.

The `page` keyword denotes a portal page. Multiple pages can be nested within a `pane` block. In each `page` block, a virtual application must be referenced using the `application` keyword. This virtual application is invoked when the portal page is displayed.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCVAPP@1.1001;
  portal CustomerPortal {
    pane OrderPane {
      colorscheme = Green;
      page OrderPage {
        application = OrderOverview;
      }
    }
  }
}
```

5 app.ducx Use Case Language

This chapter outlines use case language features that are relevant for Folio developers.

5.1 Virtual Application Context Actions

The action `FSCVAPP@1.1001:GetVAPPInformation` returns a dictionary in the first parameter, which contains the keys listed in the following table.

Note: Only for internal use.

Key	Description
`application`	The `application` key stores the virtual application that is currently running.
`dispatcher`	The `dispatcher` key stores the application dispatcher used.
`topappview`	The `topappview` key stores the topmost application view.
`topappviewisexplore`	The `topappviewisexplore` key stores whether the topmost application view is in explore mode.
`servertype`	The `servertype` key allows you to determine the type of web server used. For example, the value “IIS” is stored for Microsoft Internet Information Services.
`serverplatform`	The `serverplatform` key stores an identifier of the server operating system.
`isportlet`	The `isportlet` key stores whether it is a portlet.
`ispluginInst`	The `ispluginInst` key stores whether the native client is installed.

5.2 Extending a Virtual Application

Any virtual application can be extended with dialogs in context of a use case and variant. Use the `using` keyword followed by the reference of a virtual application and curly braces. By default, this reference is formed by concatenating the variant name and the use case name.

Syntax

```
using <reference of application> {
  dialog reference {
    ...
  }
}
```

In the following example the virtual application for the menu use case `EditInvoice` that is defined for variant `Order` gets extended by the `EditOrder` dialog.

Example

```
using OrderEditInvoice {
  dialog EditOrder {
    form = expression {
      if (coort.GetCurrentUserRoleGroup() == #SysAdm) {
        return #FormEditOrderAdmin;
      }
      else {
        return #FormEditOrderUser;
      }
    }
  }
}
```

5.3 Overriding an Existing Use Case Implementation

Syntax

```
override usecase {
  variant objectclass {
    impl = ...
  }
}
```

You can override an existing use case implementation for the object classes belonging to your software component.

When overriding an existing use case implementation, the `override` keyword must precede the reference of the use case that you want to override, followed by curly braces.

In the following example, a custom implementation is defined for `COOSYSTEM@1.1:AttrContentSet` in object class `APPDUCXSAMPLE@200.200:Invoice`.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  override AttrContentSet {
    variant Invoice {
      expression {
        // Call super method
        cooobj.CallMethod(cootx, coometh);
        if (attrdef == #content) {
          // Initialize invoice approval process
          cooobj.COOWF@1.1:InitializeWorkFlow([#APPDUCXSAMPLE@200.200:
            ProcInvoiceApproval]);
        }
      }
    }
  }
}
```

5.4 Use Case Wrappers

Syntax

```
usecase(prewrapper for ObjectCreate) {
  expression {
    ...
  }
)
```

```
usecase(postwrapper for ObjectCreate) {
  expression {
    ...
  }
)
```

Fabasoft app.ducx allows you to add a use case wrapper to an existing use case that is invoked whenever the wrapped use case is executed.

Note: Wrappers are only allowed if the software component has a 1.* domain ID (e.g. `DUCXSAMP@1.1001`).

Two types of use case wrappers are supported:

- The `prewrapper` keyword is used to assign the action as prewrapper to an existing use case. A prewrapper is invoked before the wrapped use case.
- The `postwrapper` keyword is used to assign the action to an existing use case. A postwrapper is invoked after the wrapped use case has been executed successfully.
- The parameter list is derived from the wrapped use case, modifying the parameter mode to reflect the usage of the parameters:
  - For prewrappers, input parameters become inout parameters.
  - For postwrappers, output parameters become inout parameters.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  VerifyContent(postwrapper for AttrContentSet) {
    variant Object {
      java = APPDUCXSAMPLE.Object.VerifyContent;
    }
  }
}
```

5.5 Use Case Wrappers With Data

Use case wrappers allow defining reusable building blocks with a standard implementation and polymorphism to override the default behavior.

A use case wrapper may define a prototype, a virtual application prototype, a method definition, or a virtual application.

- `COOSYSTEM@1.1:ucwprototype`
- `FSCVAPP@1.1001:ucwprototype`
- `COOSYSTEM@1.1:ucwmethdefinition`
- `FSCVAPP@1.1001:ucwapplication`

Example

app.ducx Use Case Language

```
/**
 * Default implementation for signature wrappers. If no selection is supplied,
 * it is applied to "sys_object". If "signtype" is not available for any
 * reason, an exception is thrown.
 */
usecase SignObjects(parameters as FSCVAPP@1.1001:MenuPrototype) {
  variant Object {
    application {
      expression {
        if (!sys_selobjects) {
          sys_selobjects = sys_object;
        }
        Action sys_action;
        SignatureType signtype = sys_action.signtype;
        if (!signtype) {
          throw #SIGNERR_IllegalType;
        }
        ->SignSelectedObjectsApp(sys_object, sys_action, sys_view,
              sys_selobjects, sys_selindices, sys_dynkey, signtype, null);
      }
    }
  }
}
```

app.ducx Object Model Language

```
/**
 * Defines a signature wrapper with a default virtual application
 * implementation ("SignObjects").
 */
class<UseCaseWrapper> SignatureWrapper : UseCase {
  ucwapplication = ObjectSignObjects;
  SignatureType signtype not null;
}
```

app.ducx Use Case Language

```
/**
 * "SignWithMenuInitial" creates a menu use case; it is implemented as
 * virtual application defined in "SignatureWrapper"
 */
menu usecase<SignatureWrapper> SignWithMenuInitial {
  signtype = SIGN_INITIAL;
}
```

5.6 Use Case Wrappers (Old Style)

Syntax

```
usecase {
  prewrappers = {
    prewrapper,
    ...
  }
  postwrappers = {
    postwrapper,
    ...
  }
}
```

Fabasoft app.ducx allows you to add a use case wrapper to an existing use case that is invoked whenever the wrapped use case is executed.

Note: Wrappers are only allowed if the software component has a 1.* domain ID (e.g. `DUCXSAMP@1.1001`).

Two types of use case wrappers are supported:

- The `prewrappers` keyword is used to assign one or more prewrappers to an existing use case. Multiple entries must be separated by colons. A prewrapper is invoked before the wrapped use case is executed.
- The `postwrappers` keyword is used to assign one or more postwrappers to an existing use case. Multiple entries must be separated by colons. A postwrapper is invoked after the wrapped use case has been executed successfully.

Note: A use case wrapper must be assigned the same prototype or parameter list as the use case to be wrapped by the use case wrapper.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  VerifyContent(parameters as AttrSetPrototype) {
    variant Object {
      java = APPDUCXSAMPLE.Object.VerifyContent;
    }
  }
  override AttrContentSet {
    postwrappers = {
      VerifyContent
    }
  }
}
```

6 app.ducx Organizational Structure Language

This chapter outlines organizational structure language features that are relevant for Folio developers.

An organizational structure can be represented by using abstract and concrete structure elements. These organizational structure elements can be used for assigning access rights and for defining actors in a workflow.

The following abstract elements can be used to model an organizational hierarchy independently of users and groups:

- organizational units define abstract areas of an organization (e.g. "manufacturing", "sales", "department")
- positions are used to split up organizational units into functional tasks and areas of responsibility (e.g. "production manager" in organizational unit "manufacturing", "customer representative" in organizational unit "sales").

The benefit of abstract elements is that you can model the organizational structure without considering actual users and groups. The concrete elements of the organizational structure – users and groups – can be linked to the abstract elements when your software component is deployed to the customer.

In a customer installation, groups can be linked to abstract organizational units, and users can be assigned to groups. A user can be member of one or more groups. Moreover, roles can be assigned to each user. A role is defined as the position that a user can occupy within a given group.

For instance, assume that Jane Bauer is the manager of the Sales business unit. This can be modeled by assigning a role to user Jane Bauer that is comprised of position “Manager” and group “Sales”. Furthermore, the “Sales” group must be assigned to organizational unit “Business Unit”.

Note: Users and groups are always created in the customers’ domains whereas the abstract elements, positions and organizational units, are defined using the app.ducx organizational structure language, and shipped with your software component.

6.1 Defining a Position

Syntax

```
position reference;
```

The `position` keyword is used to define a position. It must be followed by a reference and a semicolon.

Example

```
orgmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  position Clerk;
  position DeptManager;
  position DeptSecretary;
}
```

6.2 Defining an Organizational Unit

Syntax

```
orgunit reference {
  positions = {
    position,
    ...
  }
}
```

The `orgunit` keyword is used to define an organizational unit. It must be followed by a reference and curly braces.

The `positions` keyword allows you to assign positions to an organizational unit. Multiple entries must be separated by commas.

Example

```
orgmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  orgunit OrderProcessing {
    positions = {
      Clerk,
      DeptManager,
      DeptSecretary
    }
  }
}
```

6.3 Extending an Organizational Unit

Syntax

```
extend orgunit reference {
  positions = {
    position,
    ...
  }
}
```

With the `extend orgunit` keywords, you can add positions to an organizational unit that is part of another software component.

Example

```
orgmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  extend orgunit ManagementOU {
    positions = {
      DeptSecretary
    }
  }
}
```

6.4 Defining an Access Type

Syntax

```
acctype reference {
  finalform = booleanvalue;
  sequence = sequencenumber;
  symbol = symbol;
}
```

Access types are used to secure read, change and execute access to properties or use cases. If a property or use case is protected by a custom access type, the user must be granted this access type by the object's ACL in order to access the property or to invoke the use case.

Note: Software component `COOSYSTEM@1.1` already provides a set of access types that can be reused for protecting your properties and use cases.

The `acctype` keyword is used to define an access type. It must be followed by a reference and curly braces.

Within an `acctype` block, the `finalform` keyword is used to specify whether or not a user is able to access closed objects with this access type. If `finalform` is set to `false`, a user cannot access objects in final form with this access type.

The `sequence` keyword is used for defining the sequence number of the access type within the software component. If multiple access types are defined by a software component, the sequence number can be used for determining the order in which they are displayed in the ACL editor.

The `symbol` keyword is used for assigning a symbol to an access type. The symbol assigned to an access type is displayed in the ACL editor.

Example

```
orgmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  acctype AccTypeApproveOrder {
    finalform = true;
    sequence = 1;
    symbol = SymbolApprove;
  }
}
```

7 app.ducx Customization Language

This chapter outlines customization language features that are relevant for Folio developers.

7.1 Using "add" and "override" of a Software Solution or Edition (Deprecated)

If a Fabasoft Folio Domain consists of several Fabasoft Folio Tenants you might want to have an own configuration for each Fabasoft Folio Tenant. This can be achieved by defining a software solution (or software edition) for each Fabasoft Folio Tenant and using the `target` keyword followed by the reference of the software solution (or software edition) and the keyword `add` or `override`. If the keyword `add` is used, the customization is added to an existing configuration. If the keyword `override` is used, a new configuration gets generated. The configurations have to be assigned to the Fabasoft Folio Tenant manually via the Fabasoft Folio Web Client ("Domain Administration" > "Object List" > "Domain Objects").

Cloud profile note: For `EditionFolioCloud@1.1`, only `add` is allowed, `override` is forbidden for all software solutions and editions.

Example

```
// Implementation for Object
customize GetObjHint<Object> {
  // Assumes that hint is not defined as direct
  // The parameter suffix is available in the expression block
  hint = expression { cooobj.objname + " " + suffix }
}
// Implementation for ContentObject
customize GetObjHint<ContentObject> {
  // Assumes that hint is not defined as direct
  hint = expression { cooobj.objname }
}
customize GetAllowedAttrDef<Folder, objchildren> {
  // Assumes that outattrdef is defined as direct
  outattrdef = objname;
}
// Define the customization for a specific software solution
// Create a new configuration object (override)
target SolutionPayment@1.1 override {
  customize GetAllowedAttrDef<Folder, objchildren> {
    // Assumes that outattrdef is defined as direct
    outattrdef = objsubject;
  }
}
```

8 Testing and Debugging

This chapter outlines testing and debugging features that are relevant for Folio developers.

8.1 Tracing in Fabasoft app.ducx Projects

On the server, trace messages are written to the Fabasoft app.ducx Tracer, which can be found in the `Setup\ComponentsBase\Trace` folder of your installation kit.

To enable tracing, you can select several *Trace* modes in the project preferences dialog box of Eclipse.

- With trace mode "Trace Errors" enabled, errors that occur during runtime are traced.
- With trace mode "Trace Expressions" enabled, `%%TRACE`, `%%FAIL` and `%%ASSERT` directives are evaluated for your software component.
- With trace mode "Trace Calls" enabled, the invocation of each use case of your app.ducx project produces extensive trace output.

9 Sample: Creating a Wizard

The following example shows how to implement a simple wizard for creating a group.

Example

app.ducx Object Model Language

```
// The graddmemebers property is used to add members to the group
// within the second wizard step
extend class Group {
  unique Object[] graddmembers {
    allow {
      User create;
    }
  }
}
```

app.ducx User Interface Language

```
// The form contains the two form pages that are displayed in the first
// and second wizard step
form FormGroupCreateGroupWizApp {
  audience = enduser;
  formpage PageGroupCreateGroupWizApp {
    audience = enduser;
    dataset {
      grlongname;
      grshortname;
      grsupergroups;
      grorgunittype;
      objexternalkey;
    }
  }
  formpage PageMembersCreateGroupWizApp {
    audience = enduser;
    dataset {
      graddmembers;
    }
  }
}
```

app.ducx Use Case Language

```
override InitializeCreatedObject {
  variant Group {
    impl = application {
      expression {
        Object venv_object;
        Object venv_parent;
        integer venv_index;
        Object venv_view;
        Object venv_action;
        WizardContext[] @venv_wizardctx;
        Object @group;
        venv_object.ObjectLock(true, true);
        if (venv_parent.HasClass(#Group) && venv_view == #grsubgroups) {
          venv_object.grsupergroups = venv_parent;
        }
        @group = venv_object;
        // Creates the wizard context
        @venv_wizardctx = [
         {@group, #FormGroupCreateGroupWizApp, #ShowWizardApp}
        ];
        // Calls the wizard; the parameters are defined in the
        // FSCVENV@1.1001:WizardPrototype
        ->DoWizardApp(venv_object, venv_parent, venv_index, venv_view,
            venv_action, @venv_wizardctx);
        venv_object.AddUserRole(venv_object.graddmembers, #StaffPos,
            venv_object);
        venv_object.graddmembers = null;
      }
    }
  }
}
// When creating a group the implementation of InitializeCreatedObject
// gets executed
override InitializeCreatedObjectDoDefault {
  variant Group {
    // GroupInitializeCreatedObject is implicitely generated
    // from InitializeCreatedObject with variant Group
    impl = GroupInitializeCreatedObject;
  }
}
```

The wizard context is a compound property list, used to define the sequence of the wizard. The compound type `FSCVENV@1.1001:WizardContext` consists of following properties:

- `FSCVENV@1.1001:wizardobject`
  Object containing the data.
- `FSCVENV@1.1001:wizardform`
  Form or form page filtered according user profile and access check (`AccTypeSearch`).
- `FSCVENV@1.1001:wizardapplication`
  Application used for this wizard step. You can use the generic implementation `FSCVENV@1.1001:ShowWizardApp`, or create your own application with the proper prototype. Inside the application it is up to you to call `FSCVENV@1.1001:ShowWizardApp` to display the form pages.
- `FSCVENV@1.1001:wizardpageidx`
  This parameter is used internally. It stores the form page visited lastly.
- `FSCVENV@1.1001:wizardpages`
  This parameter is used internally. It stores the form pages that should be displayed depending on the access check and user profile.
- `FSCVENV@1.1001:wizardcxtvisited`
  This parameter is used internally. It stores whether this context line has been visited before.
- `FSCVENV@1.1001:wizardreadonly`
  This parameter defines whether the form pages defined in this context line should be displayed read-only.
- `FSCVENV@1.1001:wizardoptional`
  This parameter defines whether this context line and all following context lines are optional. If a context line is optional a finish branch will be displayed.

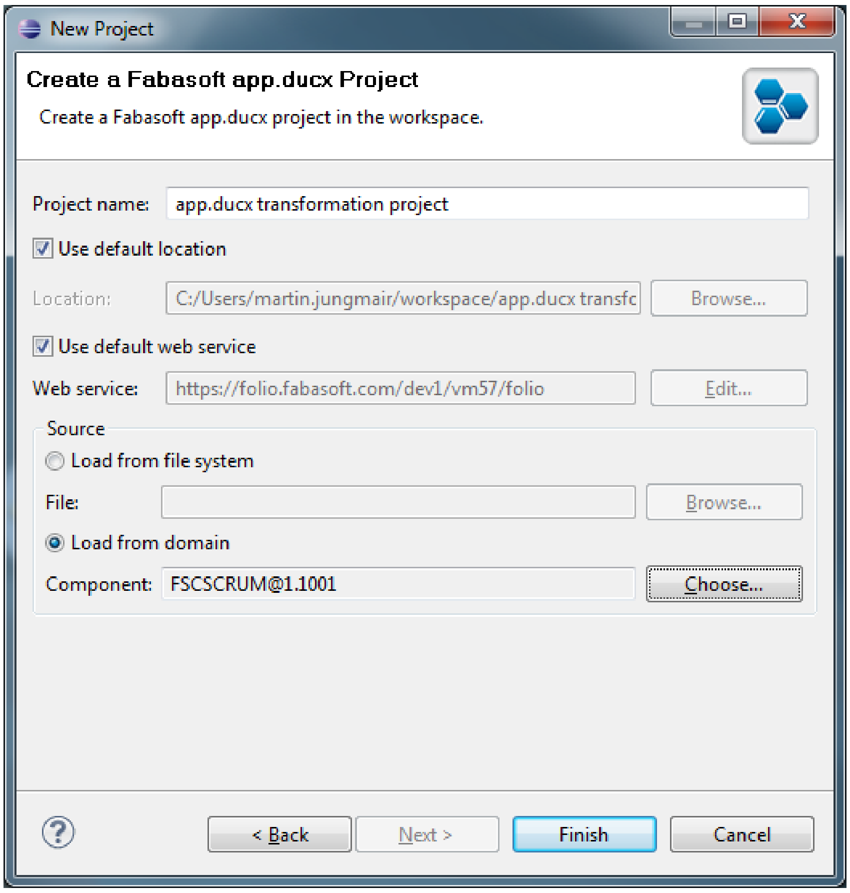
10 Transformation of Legacy Software Components

This chapter outlines the supported language constructs for the transformation of legacy software components.

10.1 Creating an Fabasoft app.ducx Project From an Existing Software Component

Fabasoft app.ducx allows you to create a Fabasoft app.ducx project from an existing software component. To do so, click “Import“ on the context menu in Eclipse Project Explorer. Select “Existing Component as Fabasoft app.ducx Project” and click “Next“.

This will take you to the dialog box depicted in the following figure where you can select the source for creating the new Fabasoft app.ducx project. You may either select a container object (`.coo`) file from the file system or a software component from your development domain. Enter a *Project name* and the reference of your new software component in the *Component* field, and click “Next“ to proceed.


New Project
Create a Fabasoft app.ducx Project
Create a Fabasoft app.ducx project in the workspace.
Source
Browse...
Edit...
Browse...
Choose...
< Back
Next >
Finish
Cancel

Project name: app.ducx transformation project

[✓] Use default location

Location: C:/Users/martin.jungmair/workspace/app.ducx transfc

[✓] Use default web service

Web service: https://folio.fabasoft.com/dev1/vm57/folio

Source: ( ) Load from file system [● Load from domain]

File:

Component: FSCSCRUM@1.1001

The remaining steps of the wizard are similar to the steps when creating a new Fabasoft app.ducx project from scratch.

After completing the remaining steps, the wizard creates a Fabasoft app.ducx project based on the software component you selected as source, and transforms the component objects belonging to the software component into domain specific language source code.

Note: A full transformation of all elements of the software component used as source might not be possible.

10.2 Object Model Language

Language Construct	Transformer Support
Class/Class extension	Supported
Struct/Struct extension	Supported
Enum/Enum extension	Supported
Instance/Instance extension	Supported
Relation	Generated as class and properties
Field	Supported

Property initialization values constraints backlink	Supported Supported Supported Supported

10.3 Resource Language

Language Construct	Transformer Support
Error message	Supported
String	Supported
Symbol	Supported

10.4 User Interface Language

Language Construct	Transformer Support
Form/Deskform dataset layout	Supported Supported Ignored
Form/Deskform extension	Supported
Formpage/Formpage extension	Supported
Menu/Menu extension	Generic Generated as an instance or a set of instances
Menu root	Supported
Portal	Generic
Taskpane/Taskpane extension	Supported
Binding	Supported
Button	Generic
Buttonbar/Buttonber extension	Supported/ Commented, when in backend is not available anymore

10.5 Use Case Language

Language Construct	Transformer Support
Transaction variable	Generated as enum and properties for `COOSYSTEM@1.1@TransactionVariables`
Usecase/Action	Supported
Override	Supported
Menu usecase	Supported
Wrappers	Generic
Virtual application	Generic

10.6 Organization Structure Language

Language Construct	Transformer Support
Position	Supported
Organizational unit/Organizational unit extension	Supported
Access type	Supported

10.7 Business Process language

Language Construct	Transformer Support
Activity/Activity extension	Supported
Process	Partially supported: conditions/loops/case construct might result errors

10.8 Customization Language

Language Construct	Transformer Support
Customization	Generic
Customization point	Generic

10.9 Expression Language

In expressions short references are used whenever possible. If parse errors occur, no changes are performed and the expression will be taken as it is.

11 Profiles

This chapter outlines the different app profiles and the respective restrictions.

11.1 Cloud App Profile Restrictions

Following restrictions apply to the cloud app profile.

11.1.1 Object Model Language

Restriction	Restricted Element	Severity

Only CompoundObject, BasicObject, ContentObject, ObjectClass, own and friend classes can be used as base classes.	Class	Error
Extending external element from non-friend component is not allowed.	Class/Enumeration/Compound Type extension	Error
Extension with address must not be used.	Class/Enumeration/Compound Type extension	Error
Edition and/or Solution instance must not be used.	Instance	Error
Application must not be instantiated.	Instance	Error
Instances with address must not be used.	Instance	Error
Not component instance must not be used.	Instance	Error
Extending external instance from non-friend component is not allowed. Exceptions: • `FSCTEAMROOM@1.1001:trchildren` • `FSCTEAMROOM@1.1001:trconfigs` • `FSCFOLIO@1.1001:EC_Root` • `COOATTREDIT@1.1:AppPackage` • Configuration instances • Instances of classes with `COOTC@1.1001:AppCategory` base class	Instance extension	Error
Configuration instances must not reference external class from non-friend component.	Configuration instance extension	Error
AppCategory instances may only assign the apps and templates properties with apps and classes from own or friend components.	AppCategory instance extension	Error
In the class chain of <instance> <class> could not be resolved. Add component <components> to project references. If this restriction fails, the other restriction evaluations on instances and instance extensions might return incorrect results.	Instance, Instance extension	Error

11.1.2 User Interface Language

Restriction	Restricted Element	Severity

External UI element from non-friend component should not be extended.	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Warning
Extension with address must not be used.	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Error

11.1.3 Use Case Language

Restriction	Restricted Element	Severity
Overriding external action from non-friend component of external classes from non-friend component is not allowed.	Overrides	Error
Wrapper must not be used.	Use case wrapper	Error
Extending external element from non-friend component is not allowed.	Dialog extension	Error
Extension with address must not be used.	Dialog extension	Error

11.1.4 Organization Structure Language

Restriction	Resticted Element	Severity
Defining organizational structure elements is not allowed.	acctype	Error
Defining organizational structure elements is not allowed.	orgunit	Error
Defining organizational structure elements is not allowed.	position	Error

11.1.5 Business Process language

Restriction	Resticted Element	Severity

Extending external element from non-friend component is not allowed.	Activity extension	Error
Extension with address must not be used.	Activity extension	Error

11.1.6 Customization Language

Restriction	Resticted element	Severity
No target add except for EditionFolioCloud@1.1 allowed.	Target definition	Error
Overriding target is not allowed.	Target definition	Error

11.2 Base App Profile Restrictions

Following restrictions apply to the base app profile.

11.2.1 Object Model Language

Restriction	Restricted Element	Severity

Only Compound-, Basic-, ContentObject, ObjectClass, own and friend classes can be used as base classes.	Class	Error
Extending external element from non-friend component is not allowed. Exception: `COOSYSTEM@1.1:UserEnvironment`	Class/Enumeration/Compound Type extension	Error
Extension with address must not be used.	Class/Enumeration/Compound Type extension	Error
Edition and/or Solution instance must not be used.	Instance	Error
Application must not be instantiated.	Instance	Error
Instances with address must not be used.	Instance	Error
Not component instance must not be used.	Instance	Error
Extending external instance from non-friend component is not allowed. Exceptions: • `FSCTEAMROOM@1.1001:trchildren` • `FSCTEAMROOM@1.1001:trconfigs` • `FSCFOLIO@1.1001:EC_Root` • Configuration instances • Instances of classes with `COOTC@1.1001:AppCategory` base class	Instance extension	Error
Configuration instances must not reference external class from non-friend component.	Configuration instance extension	Error
AppCategory instances may only assign the apps and templates properties with apps and classes from own or friend components.	AppCategory instance extension	Error
In the class chain of <instance> <class> could not be resolved. Add component <components> to project references. If this restriction fails, the other restriction evaluations on instances and instance extensions might return incorrect results.	Instance, Instance extension	Error

11.2.2 User Interface Language

Restriction	Restricted Element	Severity
External ui element from non-friend component should not be extended.	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Warning
Extension with address must not be used.	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Error

11.2.3 Use Case Language

Restriction	Restricted Element	Severity
Overriding external action from non-friend component of external classes from non-friend component is not allowed.	Overrides	Error
Wrapper must not be used.	Use case wrapper	Error
Extending external element from non-friend component is not allowed.	Dialog extension	Error
Extension with address must not be used.	Dialog extension	Error

11.2.4 Organization Structure Language

Restriction	Resticted element	Severity
Defining organizational structure model is not allowed.	Organizational structure model	Error

11.2.5 Business Process language

Restriction	Resticted element	Severity

Defining positions and/or organizational units for activities is not allowed.	Activity	Error
Extending external element from non-friend component is not allowed.	Activity extension	Error
Extension with address must not be used.	Activity extension	Error

11.2.6 Customization Language

Restriction	Resticted element	Severity
Adding customizations to a target other than EditionFolioCloud@1.1 is not allowed.	Target definition	Error
Overriding target is not allowed.	Target definition	Error

11.3 Enterprise App Profile Restrictions

Following restrictions apply to the enterprise app profile.

11.3.1 Object Model Language

Restriction	Restricted Element	Severity

Only Compound-, Basic-, ContentObject, ObjectClass, own and friend classes can be used as base classes.	Class	Error
Extending external element from non-friend component is not allowed.	Class/Enumeration/Compound Type extension	Error
Extension with address must not be used.	Class/Enumeration/Compound Type extension	Error
Edition and/or Solution instance must not be used.	Instance	Error
Application must not be instantiated.	Instance	Error
Instances with address must not be used.	Instance	Error
Not component instance must not be used.	Instance	Error
Extending external instance from non-friend component is not allowed. Exceptions: • `FSCTEAMROOM@1.1001:trchildren` • `FSCTEAMROOM@1.1001:trconfigs` • `FSCFOLIO@1.1001:EC_Root` • Configuration instances • Instances of classes with `COOTC@1.1001:AppCategory` base class	Instance extension	Error
Configuration instances must not reference external class from non-friend component.	Configuration instance extension	Error
AppCategory instances may only assign the apps and templates properties with apps and classes from own or friend components.	AppCategory instance extension	Error
In the class chain of <instance> <class> could not be resolved. Add component <components> to project references. If this restriction fails, the other restriction evaluations on instances and instance extensions might return incorrect results.	Instance, Instance extension	Error

11.3.2 User Interface Language

Restriction	Restricted Element	Severity

External ui element from non-friend component should not be extended.	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Warning
Extension with address must not be used.	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Error

11.3.3 Use Case Language

Restriction	Restricted Element	Severity
Overriding external action from non-friend component of external classes from non-friend component is not allowed.	Overrides	Error
Wrapper must not be used.	Use case wrapper	Error
Extending external element from non-friend component is not allowed.	Dialog extension	Error
Extension with address must not be used.	Dialog extension	Error

11.3.4 Business Process language

Restriction	Resticted element	Severity
Defining positions and/or organizational units for activities is not allowed.	Activity	Error
Extending external element from non-friend component is not allowed.	Activity extension	Error
Extension with address must not be used.	Activity extension	Error
The use of wait elements is not allowed.	Process	Error

11.3.5 Customization Language

Restriction	Resticted element	Severity
No target add excepting for EditionFolioCompliance@1.1 allowed.	Target definition	Error

Overriding target is not allowed.	Target definition	Error