What’s New

Fabasoft app.ducx

Fabasoft

Contents

1 Fabasoft app.ducx 2017 June Release

Find out more about new features and improvements in the Fabasoft app.ducx 2017 June Release.

1.1 New Keyword “assume“

In expressions a new way of (re)declaring variables is possible using the keyword `assume`. The app.ducx compiler does not create a declaration for these variables in the resulting kernel expression, but it creates a few assertions regarding the presence and the dynamic value of the variable.

Redefine a variable

`assume` can be used to redefine the type of already available variables or parameters.

Example

```
usecase OnUserAndGroup(any memberof) {
  variant User, Group {
    expression {
      if (cooobj.HasAttribute(cootx, #usersurname)) {
        assume User coooobj;
        assume Group memberof;
      }
    }
  }
}
```

Define a variable

Sometimes it is necessary to access variables which are available in a given scope but the app.ducx compiler does not know about them. This is very often the case in application implementations.

These can be declared by writing something like an inline parameter declaration using the keyword `assume`.

Example

```
usecase SetReturnURL() {
  variant User {
    application {
      expression {
        assume in string sys_branchvalue;
        assume out ::ru;
        …
      }
    }
  }
}
```

1.2 Overview of Free Addresses

The address range drop-down field in the properties dialog now shows the free addresses per file.

1.3 Edit/Synchronize Simplified

As there are more and more keywords for complex constructs like ACLs and doc properties, the edit/synchronize functionality has been reworked and is currently only available in a very basic form. Features will be added to this functionality in the future.

1.4 Import of Projects from COO Files

Importing projects from COO files is no longer supported (as files and from the server).

2 Fabasoft app.ducx 2017 February Release

Find out more about new features and improvements in the Fabasoft app.ducx 2017 February Release.

2.1 Contexts in Customization Language

Some customization points support additional configuration contexts. This context is calculated dynamically by calling the action `FSCCCONFIG@1.1001:SetCPContext(object cp, retval object context)` on the object the customization point is called for.

Example

```
// Define the customization for the software edition Folio
// Add an entry to an existing configuration object in the Folio configuration,
// additionally specifying a context
target DomainTypeFolio.Art {
  customize GetAllowedAttrDef<Folder, objchildren> {
    outattrdef = objsubject;
  }
}


target DomainTypeFolio.[Art,Business] {
  customize GetAllowedAttrDef<Folder, objchildren> {
    outattrdef = objsubject;
  }
}
```

Contexts can be listed in the target definition.

It is only allowed to configure the contexts, when one of the following conditions hold true:

- the domain type configuration object belongs to the current project or
- the customization point belongs to the current project or
- the context object belongs to the current project

2.2 Expressions in Customizations

Local scope, global scope and parameters are considered in all expressions used in customizations.

2.3 Parameter of Customizations Checked

Only key parameters can be used as parameters in customizations.

2.4 Attribute Definition Class

If a developer needs to manually define the class of property, she can do that by explicitly specifying the property `COOSYSTEM@1.1:attrtype` in an attribute definition. We have implemented a check of the base class to ensure correct results.

2.5 Extending an Access Control List

Syntax

```
extend acl reference {
  ace {
    audience = {
    };
    rights = [];
  }
}
```

With the `extend acl` keywords, additional `ace`s can be added to existing access control lists.

2.6 Customization Projects

By simply checking a checkbox the app.ducx project is turned into a customization project. Such projects do not create software components and component objects. Instead they contain only subtle changes or additions to existing software components.

2.7 # Operator

Accessing properties of variables now result in warnings, if the identifier is ambiguous. By using the `#` operator the resolving as full reference can be forced.

Example

```
// Accessing property of undeclared object
@undeclaredobj.#objname = "New Name";
// Accessing key of undeclared dictionary
@undeclareddict.$objname = "New Name";
```

2.8 -> Operator

The invocation of applications using objects and the detach operator is checked now. There are warnings when there is no application mapping, when there are no or wrong parameters and other checks.

2.9 cooobj in Applications

The variable `cooobj` in applications implementing actions for a specific object class is a variable with the type of the implemented object class now.

3 Fabasoft app.ducx 2016 December Release

Find out more about new features and improvements in the Fabasoft app.ducx 2016 December Release.

3.1 Quick Fix for Property Actions

When specifying not implemented actions as triggers in a property definition, the quick fix for this action supplies the correct parameter prototype and object class in the generated use case.

3.2 Quick Fix for Display Views

We have implemented transformer from generic display views to `columns`, including `mlname` format conversion.

3.3 Sorting for Column Groups

In column groups the sorting direction can now be specified.

Example

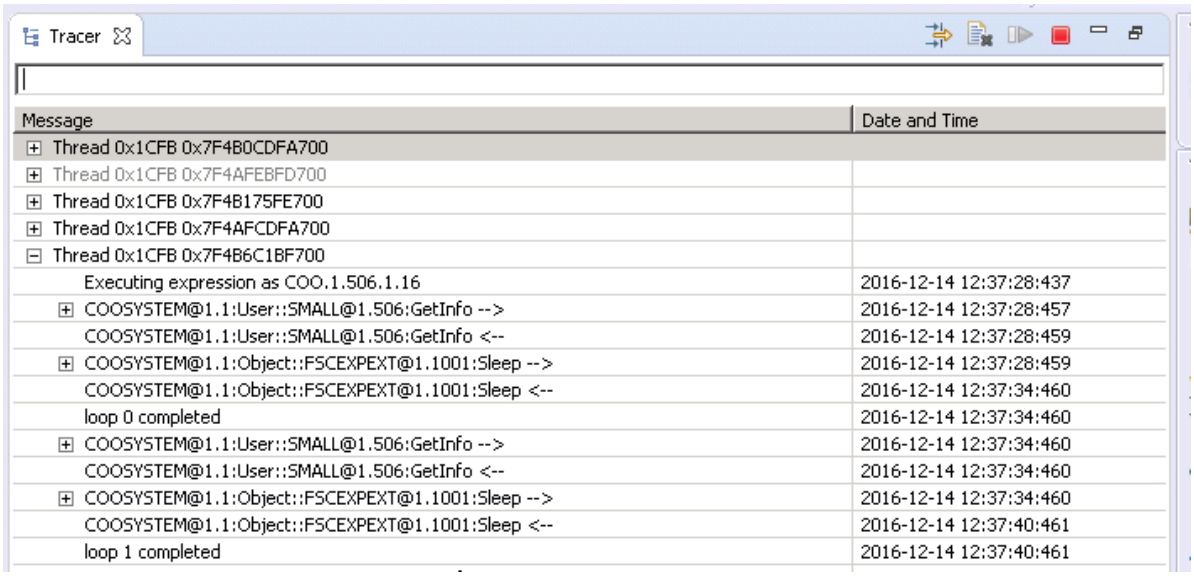
```
columns for Folder (
  objchildren {
    objname;
    objsubject;
    objclass;
  }
  groupby {
    objclass asc;
  }
)
```

3.4 Tracer View

A different view to the trace is provided by the Tracer View. The trace flags in Preferences are also used for this view.

This view contains buttons to start and stop the tracer. If not running while executing a expression it is automatically started and after completion of the expression stopped.

The Tracer View maintains tree nodes for every executing thread.



4 Fabasoft app.ducx 2016 September Release

Find out more about new features and improvements in the Fabasoft app.ducx 2016 September Release.

4.1 Cloud Profile Restrictions

The restrictions for organization models as whole have been changed to restrictions for single elements in the organizational language.

Restriction	Restricted Element	Severity
Defining organizational structure model is not allowed.	`Acctype`	Error
Defining organizational structure model is not allowed.	`orgunit`	Error
Defining organizational structure model is not allowed.	`position`	Error

4.2 Defining a Prototype

If there are multiple use cases with the same parameters, it is possible to define the parameters once as prototype with the keyword `prototype`. A prototype specifies the arguments of a use case, including the type, the name and the in/out settings.

Example

```
prototype AttrSetDatePrototype (
  AttributeDefinition attrdef,
  ref datetime value,
  datetime oldvalue
);
```

There is also the possibility to specify expression parameters with the key words `expression prototype`.

Example

```
expression prototype boolean LogPrototype (
  string data,
  out string line
)
```

4.3 Defining a Method Definition

To supply just an implementation without use case mapping, the keyword `method` can be used.

Example

```
method MDUpdateOwner {
  impl = expression {
    cooobj.ObjectLock(true, true);
    cooobj.objowner = coouser;
  }
)
method MDEmpty {
  impl = empty;
  }
)
```

4.4 Defining Wrappers

The execution engine supports injection of actions before or after any action. This is implemented with app.ducx by using the keyword `prewrapper` and `postwrapper` inside the parameter list of an `action` or `usecase`.

The given action is called before or after the wrapped one, providing a way to modify the result of the action call.

Example

```
action ObjectCreateWrapper(postwrapper of ObjectCreate) {
  impl = expression {
    cooobj.objname = "Created at" + coonow;
  }
)
```

4.5 Defining Display Views

The element `displayview` in `ducx-ui` transforms column information into objects of the object class `COOSYSTEM@1.1:DefaultDisplayView`.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  displayview UserView {
    // Expression property
```

```
    columns objsecuser {
      freeze {
        objfirstname;
        objsurname;
      }
      objcreatedat;
      groupby {
        objcreatedat day;
      }
      sortby {
        objsurname;
        objfirstname desc;
      }
      displaymodes {
        LISTVIEW_DETAILS
      }
    }
  }
}
```

4.6 Extensions to Expression Properties

An expression property now can have additional information regarding the evaluation context.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class StatusLog : BasicObject {
    expression boolean logexpr(string data, out string line) {
      scope = PARSCOPE_GLOBALSCOPEWITHOBJECT;
  }
}
```

This expression property describes an expression with the following behavior:

- It operates on an object.
- It gets a string value `data` in the global scope.
- It creates a string value `line` in the global scope.
- It returns `true` or `false`.

4.7 Assigning Default Column Settings to an Object Class

When displaying object lists in the web client, the default columns should be specified in the object class. This can now be accomplished by using the keyword `columns`.

Syntax

```
extend class reference {
  columns attrdef {
    freeze {
      column1;
      column2;
```

```
    }
    ….
    groupby {
      column1;
    }
    sortby {
      column2 desc;
      …
    }
    displaymodes {
      modes
    }
  }
}
```

4.8 Reference Documentation

Comments starting with `/**` are used to create a reference documentation for the component object behind the comment. This follows the Javadoc recommendations.

Additionally there is the possibility to tag source code with `<expr>` and `</expr>`. Code will be formatted with a fixed space font.

```
/**
 * customization point for any list
 * <expr>
 * customize abc<Object> {
 *   users = expression {
 *     return [coouser];
 *   }
 * }
 * </expr>
 */
CPValue(out User[] users);
```

SMALL@1.506:CPValue(out User[] users)

customization point for any list

```
customize abc<Object> {
  users = expression {
    return [coouser];
  }
}
```

4.9 Kernel Constants coogroup and cooposition

There are new predefined constants for the role of the current user:

- The `coogroup` variable holds the group object of the role of the current user.
- The `cooposition` variable holds the position object of the role of the current user.

4.10 Stronger Checks

The compiler has been extended by the following additional checks:

- Terms in `if`, `while` and `for` statements must be `Boolean`.
- Terms in `case` statements must match the term in the `switch` statement.
- All elements of arrays must have the same type.
- In the expression implementation of use cases the variable `cooobj` has the correct object class.
- Kernel Interface calls return the correct object class.

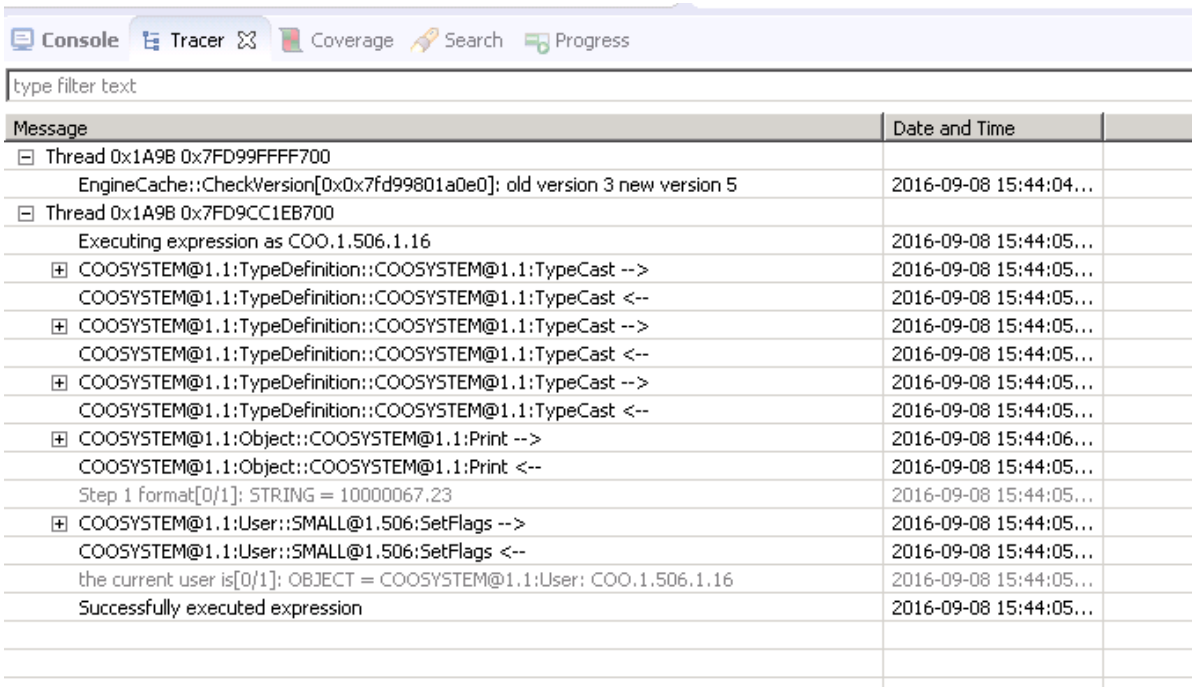
4.11 Expression Tester Improvements

4.11.1 Shortcut Alt + X

The shortcut for firing any expression is now `Alt + X`.

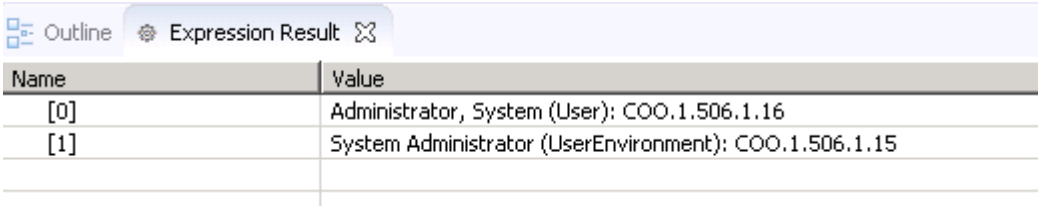
4.11.2 Tracer

When evaluating an expression, a tracer session is automatically started and the “Tracer” view is displayed. This view works asynchronously and collects all traces until the expression is completed. This view orders all traces by threads and is especially aware of the call stack information.



4.11.3 Result Windows

The result of the expression evaluation is displayed in an “Expression Result” view.



4.12 Java Tracing

When compiled without tracing, now calls of the built-in functions `DUCX.trace`, `DUCX.traceEnter` and `DUCX.traceLeave` are removed from the Java byte code, too.

4.13 Ant Task coverage-persist

The ant library has been extended by a task to persist the running coverage session.

4.14 Kernel Interfaces

Handling of Kernel Interfaces has been improved extensively.

- Parameters and result types are considered in expressions.
- Kernel constants are known to the editor.
- Tooltips for all interfaces is displayed.
- Code completion for kernel flags is implemented.
- Kernel constants are marked as keywords.

4.15 Extending Component Objects with Aggregates

In the case of a software component member in a new aggregate line, this member is initialized automatically with the current software component. The member has to be set by the developer only if the software component is a different one.

4.16 Find References

The context menu entry “Find References” with the shortcut `Ctrl + Shift + G` finds all references of an entity in the project.

4.17 Menu Bindings Syntax

To reduce indentation levels, a new syntax for menu bindings has been introduced.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  import FSCTEAMROOM@1.1001;

  menus for Package {
    TaskpaneSelectedExpansion, MenuContextExpansion.objchildren {
      priority = 100;
      MenuEncryptObject {
        condition = expression { !cooobj.isencrypted }
```

```
      }
      MenuDecryptObject {
        condition = expression { cooobj.isencrypted }
      }
    }
    scope DomainTypeFolioCloud {
      MenuContextExpansion.trchildren {
        priority = 100;
        MenuEncryptObject {
          condition = expression { !cooobj.isencrypted }
        }
        MenuDecryptObject {
          condition = expression { cooobj.isencrypted }
        }
      }
    }
  }
}
```

There are a few new features implemented:

- After the `for`-statement one or more object classes are listed.

- The menu block can be surrounded with a scope block
- A target can be specified by adding "." And the reverence of the view to the expansion point
- Menu items can be listed directly
- In menu item blocks there can be some generic assignments. These override the assignments in the outer block

4.18 Menu Binding Names

The recommended names of the menus have been harmonized and aliases for the task panes have been added:

Keyword	Description
`mainmenu`	With the `mainmenu` keyword, a menu root can be assigned to an object class that is used as its main menu.
`contextmenu`	With the `contextmenu` keyword, a menu root can be assigned to an object class that is used as its context menu.
`windowmenu`	With the `windowmenu` keyword, a menu root can be assigned to an object class that is used as its background menu.
`independentmenu`	With the `independentmenu` keyword, a menu root can be assigned to an object class that is used as its container independent menu.
`tasks`	With the `tasks` keyword a taskpane can be assigned to an object class.
`selectedtasks`	With the `selectedtasks` keyword a taskpane can be assigned to an object class that contains menu entries of the selected objects.

4.19 Form Bindings Syntax

To reduce indentation levels, a new syntax for form bindings has been introduced.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  import COOATTREDIT@1.1;
  import COOSEARCH@1.1;
  import COOTC@1.1001;
  forms for Order {
    OpenObject, ReadObjectAttributes, EditObjectAttributes {
      OrderAdminForm;
      OrderUserForm;
    }
    SearchObjects {
      OrderSearchForm;
    }
    ObjectConstructor {
      OrderCreateForm;
    }
  }

}
```

5 Fabasoft app.ducx 2016 June Release

Find out more about new features and improvements in the Fabasoft app.ducx 2016 June Release.

5.1 Keyword “obsolete” in Enumeration Types

The `obsolete` keyword can be used to disable enumeration entries. In the user interface, drop-down lists for enumeration values do not contain disabled entries.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  // Enumeration type consisting of four enumeration items with
  // explicitly assigned integer values, some of them disabled
  enum ShipType {
    ST_DESTROYER = 100,
    ST_CRUISER = 200,
    obsolete ST_BATTLESHIP = 300,
    obsolete ST_SUBMARINE = 400
  }
}
```

In rare cases it is necessary to specify some other attributes for the type. This is accomplished by using generic assignments before the block of enumeration values.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  // Enumeration type consisting of four enumeration items with
  // explicitly assigned integer values and some generic assignments before
  enum ShipType {
    typemultiple = true;
    typelistunique = true;
    ST_DESTROYER = 100,
    ST_CRUISER = 200,
```

```
    ST_BATTLESHIP = 300,
    ST_SUBMARINE = 400
  }
}
```

5.2 Defining an Expression Property

An expression property is a string list property that stores the expression and has additional information regarding the evaluation context.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class StatusLog : BasicObject {
    // Expression property
    expression logexpr {
      scope = PARSCOPE_OBJECT;
      type = BOOLEAN;
      parameters<actparname, actparmod, actpartype, actparretval> = {
         { "info", PARMOD OUT, STRING, true }
      }
  }
}
```

5.3 Defining an Access Control List

Syntax

```
acl reference {
  ace {
    audience = {
    };
    rights = [];
  }
}
```

Access control lists (ACLs) are used to specify the access rights of a user to a given object. An access control list can assign different lists of access types to different user groups.

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

Within an `acl` block, there is a sequence of `ace`, specifying a unique set of access rights to a list of user groups, called `audience`. In the resulting access control list there is a line for each user group and the specified access types.

Each audience entry can define something that depends on the user, on a group the user belongs to and on the domain where the user is located. These three possibilities of specifying the user are modelled with the keywords `user`, `group` and `domain`. For each of these keywords, there are different possibilities to define the matching user group.

If one or more of the keywords are omitted, the line is filled up with default values.

Example

```
orgmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  acl SampleACL {
```

```
    ace {
      audience = {
         user SysAdm;
      };
      rights = [AccTypeRead, AccTypeChange];
    }
    ace {
      audience = [
        {
           user ACLUSER_DEFAULT;
           group ACLGROUP_OWNER if parent;
           domain ACLDOMAIN_DEFAULT;
        },
        {
        }
      ];
      rights = [AccTypeRead];
    }
  }
}
```

Audience elements can also be declared as constants and reused in all ACLs.

Example

```
orgmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;

  const audience[] AdministrationAudience = {
    user = SyAdm;
  }

  acl SampleACL {
    ace {
      audience = AdministrationAudience;
      rights = [AccTypeRead, AccTypeChange];
    }
  }
}
```

5.4 Expression Tester

The expression tester works on all files in the project, either for expression files with the extension `ducx-xp` or inside of any expression element.

The expression can be executed by the current web service by selecting “Run as” > “Expression” from the context menu.

The result of the expression is sent to the expression tester console.

To use short references imports have to be done as in all other domain specific files, the context is the current project.

5.5 Extending Properties

Extending properties can now be done by using the keyword extend, followed by the property. Using this syntax, object pointer properties can be extend in a simple way.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  import COOATTREDIT@1.1;
  import COOSEARCH@1.1;
  import COOTC@1.1001;
  extend property admobjchildren {
    allow {
      MySimpleClass;
    }
  }
}
```

5.6 Multiple Use Cases in Form Assignments

The `forms` binding can now specify more than one use case in a single binding block.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  import COOATTREDIT@1.1;
  import COOSEARCH@1.1;
  import COOTC@1.1001;
  extend class Order {
    forms {
      OpenObject, ReadObjectAttributes, EditObjectAttributes {
        OrderAdminForm;
        OrderUserForm;
      }
      SearchObjects {
        OrderSearchForm;
      }
      ObjectConstructor {
        OrderCreateForm;
      }
    }
  }
}
```

5.7 Conditions in Form Assignments

The condition in `forms` bindings can now specified by using the condition keyword.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  import COOATTREDIT@1.1;
  import COOSEARCH@1.1;
  import COOTC@1.1001;
  extend class Order {
```

```
    forms {
      OpenObject, ReadObjectAttributes, EditObjectAttributes {
        OrderAdminForm {
          condition = expression {
            coort.GetCurrentUserRolePosition() == #SysAdm;
          }
        }
        OrderUserForm;
      }
    }
  }
}
```

5.8 Additional Quick Fixes

There are some additional quick fixes available:

- Quick fix for cast from multiple to single
- Quick fix to declare variables with the correct type

5.9 Additional Object Model Templates

There are some additional templates for the object model:

- New operation in a new web service
- New operation in an existing web service
- New friendly URL

5.10 Additional Warnings

There are some additional warnings generated:

- `Found` cyclic `dependency between <swca> and <swcb>`
  This warning is issued if an added software component has dependencies to the current software component.
- `Invalid assignment: should not set property <property>`
  This warning is issued if assignments to component objects cannot be marked as changes of the current software component. These are either simple values or assignments of other software components.

5.11 Additional Errors

There are some additional errors generated:

- `Resolving: <class> <object> cannot be used. Missing component: <swc>`
  This error message is issued if there is not enough information to check the contents of an object pointer property because of a missing software component

5.12 New Formatter

We use a new formatter to format the source code. The new features include

- better performance
- increased flexibility with fixed syntax elements
- conditional formatting

- tabular formats

6 Fabasoft app.ducx 2016 April Release

Find out more about new features and improvements in the Fabasoft app.ducx 2016 April Release.

6.1 Syntax Changes

Some syntax changes have been introduced to harmonize the object model syntax with the expression syntax.

References with "#"

In simple assignments of component objects a hash can be used as marker for a component object. Currently the hash is ignored but may be used for additional context checks in the future.

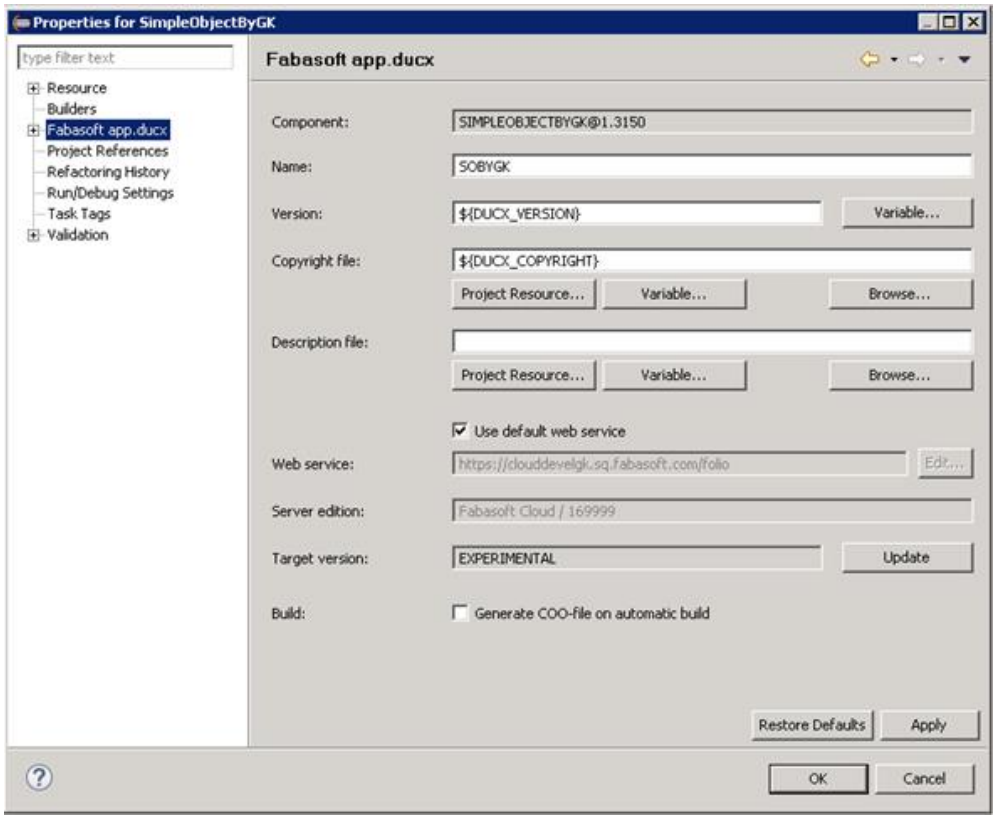
Array initializers

Arrays in object model assignments now use the comma as separators instead of the semicolon.

6.2 Target Version

The compiler generates code for a specific target version. This version is stored in the app.ducx project file and is not changed anymore by just connecting to a web service. The properties dialog of projects display the version of the currently used web service and the specified target version.

By clicking the "Update" button the target version of the current web service is stored in the project file and subsequently used for the compiler.



6.3 Generated Files

Object address as comment

The generated file now contains the object address of extended objects as comments.

attrmultiple/actparmultiple

Attributes and parameters are explicitly declared as multiple when using a multiple type definition.

6.4 Expression as Type for Attributes

It is now possible to declare an expression property by using the type expression. Additionally the useful properties of an expression property are defined as aliases.

Example

```
expression myexpression {
  type = BOOLEAN;
  scope = PARSCOPE_LOCALSCOPE;
  parameters = [
    {
      actparname = "info",
      actpartype = #STRING,
      actparmod = PARMOD_OUT,
      actparretval = true
    }
  ];
}
```

6.5 New Language Element “constant”

In all files it is now possible to declare constant elements.

Example

```
private enum DrinkType {
  DT_SHOT = 800,
  DT_LONGDRINK = 850,
  DT_SOUR = 851,
  DT_FIZZ = 852,
  DT_HIGHBALL = 853,
  DT_SOFTDRINK = 1000,
  DT_WATER = 1001,
  DT_JUICE = 1002
}
struct SomeLines {
  string line1;
  string line2;
  string line3;
}
class Bar : ComponentObject {
  DrinkType[] barinitwith;
  DrinkType[] barinitwithout;
  SomeLines barinfo;
}
const DrinkType[] AlcoholicDrinksConst = [DT_SHOT, DT_LONGDRINK, DT_SOUR, DT_HIGHBALL];
const DrinkType[] NonAlcoholicDrinksConst = [DT_SOFTDRINK, DT_WATER, DT_JUICE];
const SomeLines LineConst = {
  line1 = "First 1",
  line2 = "Second line"
}
const SomeLines[] LinesConst = [
```

```
  {
    line1 = "First 1",
    line2 = "Second line"
  },
  {
    line1 = "First 1",
    line2 = "Second line"
  },
  {
    LineConst,
    line2 = "Second line override",
    line3 = "Third line"
  }
];
```

The first use of these constants is in object model assignments:

Example

```
const SomeLines LineConstEx = {
  LineConst,
  line2 = "Second line override",
  line3 = "Third line"
}
instance Bar MyDefaultBar {
  barinitwith = AlcoholicDrinksConst;
  barinitwithout = [DT_WATER, NonAlcoholicDrinksConst];
  barinfo = {
    LineConstEx,
    line1 = "First line from init"
  }
}
```

The other use is in expressions, where the constant is replaced by its text like a macro:

Example

```
instance Expression ex {
  exprtext = expression {
    DrinkType aDrink = DT_FIZZ;
    if (aDrink in AlcoholicDrinksConst) {
      coort.Trace("this drink contains alcohol", aDrink);
    }
    else if (aDrink in NonAlcoholicDrinksConst) {
      coort.Trace("this drink does not contain alcohol", aDrink);
    }
    else {
      coort.Trace("this drink might contain alcohol", aDrink);
    }
    SomeLines myline = {
      LinesConst,
      line3 = "a new line"
    };
    SomeLines[] mylines = [
      LinesConst,
      {
        line3 = "a new line"
      }
    ];
    mylines = LinesConst;
  }
}
```

If the target version supports the object class `COOSYSTEM@1.1:TypedExpression`, these constants are stored as component objects and can be used by other software components, too.

6.6 Quick Fix for Missing Arguments

The warning for missing arguments due to the declaration of the called use case or application now contains quick fixes to add the missing argument to the call.

7 Fabasoft app.ducx 2016 February Release

Find out more about new features and improvements in the Fabasoft app.ducx 2016 February Release.

7.1 Special Layout Element <static>

To display a static text on a form page, the new keyword `static` has been introduced. By using this keyword it is quite easy to bring a static text, provided by a string object, on a form page.

Example

```
formpage PageOrgIndustry {
  audience = enduser;
  dataset {
    orgindustry;
  }
  layout {
    row {
      static StrTxt {
        colspan = 2;
      }
    }
  }
}
```

Note: The defined text is embedded directly as HTML on the form page. Simple HTML elements can be used to format the text. The text is not escaped; therefore pay attention to the used HTML.

7.2 Authentication via Apache Ant with Client Certificates

It is now possible to authenticate a user via Apache Ant using client certificates.

The web service task has been extended to specify a client certificate and a trust store.

Client Certificate Example

```
<target name="definewebsvc">
  <webservice id="websvc" url="your baseurl" timeout="your timeout">
    <authentication>
       <clientcertificate>
         <keystore location="/ssl/keystore.p12" password="xxx" type="pkcs12" />
         <truststore location="/ssl/truststore.jks" password="xxx" type="jks" />
       </clientcertificate>
    </authentication>
  </webservice>
</target>
```

7.3 Specifying a Concrete Compound Type

Sometimes it is necessary to use a more specialized compound type as type for a compound property. This can be achieved by denoting the required type after the `struct` keyword.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  struct<TypeCustomizationPointDef> CPManualCreated {
    ObjectClass objcls;
    component;
    string title;
  }
}
```