ArchGenXML - Getting started

ArchGenXML is a code-generator for CMF/Plone applications (Products) based on the Archetypes framework. It parses UML models in XMI-Format (.xmi, .zargo, .zuml), created with applications such as ArgoUML, Poseidon or ObjectDomain. This tutorial will help you get started developing applications with the aid of ArchGenXML.

jensens

Contents

Introduction
An introduction to ArchGenXML - what it is, reasons to use it, who made it.

Installation
How to install ArchGenXML and get up and running.

UML
A brief introduction to UML and pointers to further readings.

Getting started
Create your first content type with ArchGenXML

Basics: Classes / Content-Types
How to generate content types, tools and interfaces.

Basics: Attributes / Fields
How to control the fields of your schema.

Basics: Widgets
Simple content type creation: Setting up the Widgets.

Basics: Methods and Actions
Simple Content-Type creation: Defining Methods and Actions

Basics: Relationships between classes and objects
How to use references, associations, aggregations and compositions

Basics: Workflow Generation
ArchGenXML can use state diagrams to create custom workflows for your types.

Third Party Product Integration: ATVocabularyManager

ATVocabularyManager is a product for letting site managers define vocabularies for fields through-the-web or by import from XML files.

ArchGenXML can generate the necessary code to use this product.

Third Party Product Integration: Relations

Create relations between portal-types model-driven. Support for Relations Product (complex references). Define sets of rules for validation, creation and lifetime of Archetypes references. ArchGenXML can generate the necessary code and XML-configuration data to use this product.

Quick Reference

A very nice Quick Reference for ArchGenXML abusers.

Tagged value overview

This pages provides the full overview of the tagged values that are recognised by ArchGenXML.

Stereotype overview

Overview of all stereotypes you can use. (Note 2005-08-25: stereotypes are complete, but there are some empty descriptions). This pages is autogenerated with ‘python UMLProfile.py’.

Introduction

An introduction to ArchGenXML - what it is, reasons to use it, who made it.

What is ArchGenXML

With ArchGenXML you can create working python code without writing one single line of python. It is is a commandline utility that generates fully functional Zope Products based on the Archetypes framework from UML models using XMI (.xmi, .zargo, .zuml) files. The most common use case is to generate a set of custom content types, possibly with a few tools, a CMFMember type and some workflows thrown in.

In pratice, you draw your UML diagrams in a tool like Poseidon or ObjectDomain which has the ability to generate XMI files. Once you are ready to test your product, you run ArchGenXML on the XMI file, which will generate the product directory. After generation, you will be able to install your product in Plone and have your new content types, tools and workflows available.

At present, round-trip support is not implemented: Custom code can't be converted back into XMI (and thus diagams). However, you can regenerate your product over existing code. Method bodies and certain "protected" code sections will be preserved. This means that you can evolve your product's public interfaces, its methods and its attributes in the UML model, without fear of losing your hand-written code.

ArchGenXML is hosted at svn.plone.org as a subproject of the Archetypes project. It is released under GNU General Public Licence 2 or later.

Why should I use ArchGenXML?

Major reasons:

- You want to save time
- You are a lazy programmer
- You don't like to reinvent the wheel
- You dont like copying and pasting code and bugs
- You make heavy use of references and interfaces
- You have big projects with many different custom types
- You want or need a well-documented interface to your product
- You like structured model- and pattern-driven software development
- You want to maintain your project in future without getting a headache

and many more good and odd other reasons.

Contributors

The project was initially started by Phil Auersperg. Thanks to his laziness :-)

Authors

Phil Auersperg (Project Leader)

BlueDynamics GmbH, phil@bluedynamics.com,

Jens Klein (Developer and Doc-Writer)

jens quadrat, Klein & Partner KEG, jens.klein@jensquadrat.com,

Fabiano Weimar dos Santos (Ideas, Testing, Bugfixing, Workflow)

Weimar Desenvolvimento e Consultoria em Informatica Ltda., xiru@xiru.org,

Martin Aspeli (Improvements, bug fixes and documentation)

Martin Aspeli

and others

thanks to everybody who contributed
with testing, doc-writing or code-pieces!

Sponsors

Xiru.org, Brazil (Fabiano Weimar dos
Santos) sponsors a valuable amount
of money into workflow support
(State diagrams -> DCWorkflow,
will go into release 1.2),

PilotSystems, Paris, France (David Sapiro),

OpenSource.ag, Innsbruck,
Austria (Georg Pleger).

If you want to contribute ArchGenXML by improving the code, helping with documentation or sponsoring money to make us improve it, please contact one of us.

Installation

How to install ArchGenXML and get up and running.

ArchGenXML

Preconditions

You will need a working Python interpreter, version 2.3 or later.

You will need Plone 2 installed (choose the latest stable release) and its dependencies to see your generated code in action.

We also recommend to upgrade Archetypes to the latest stable release, preferrably 1.3.x or later.

Download

You need to download the release tarball of ArchGenXML from Sourceforge. You'll find it in the `files` area of the Archetypes-project. Choose the most recent version or use the bleeding edge development version from the Subversion repository.

Installation

Simply un-tar the downloaded file to a directory of your choice and remember the path to ArchGenXML.py. You **do not** have to put it in Zope's Products directory!

If you are running on a unix-like operating system, we suggest you give the file execution permissions and make a symbolic link at a place mentioned in your `PATH` environment variable. That way, you can execute ArchGenXML simply with the command `ArchGenXML.py`.

Note: The installation will be handled by dist-utils in one of the next releases, which should make it a lot easier. :-)

Additional software

To get all the features of ArchGenXML, you may need some of the following.

For code generation:

i18ndude

Without this, the generation of translatable user interface strings is disabled.
Download and install i18ndude from the plone-i18n project on Sourceforge.net.

Stripogram

Some UML tools produce HTML in the documentation elements in XMI. Stripogram converts them into plain text. Without having Stripogram installed this feature is disabled. Download and install stripogram from the squishdot project on sourceforge.net.

For running the generated code

ATVocabularyManager

Enables usage of custom dynamic vocabularies. Download and install ATVocabularyManager directly from Archetypes SVN.

UML Tools

ArchGenXML processes models stored in XMI. This XML format isn't intended to be written in a plain text editor nor in a tree based XML editor, so you will almost certainly use a UML design tool. Below is a more or less complete list of such tools. If you know about any others tools missing from this list, have more detailed information or have experience with a tool in combination with ArchGenXML, please write the author a short e-mail.

Poseidon (by Gentleware)

Website and download: www.gentleware.com,

Do **not use Version 3.1***, there are problems with reading of old models and generation of workflows!* **Use 3.0.x instead.**

Commercial software - Community Edition freely available, supports XMI version 1.2

Written in Java, runs on most platforms

Based on ArgoUML

Stores the model natively as XMI + diagram information in .zuml files (zip files)

Is very slow

Needs lots of memory and a fast CPU

This author's preferred choice

ArgoUML

Website and download: argouml.tigris.org

Free software

Written in Java

- Runs on most platforms
- Stores the model natively as XMI + diagram information in .zuml files (zip files)
- Some known, but non-critical bugs

ObjectDomain

- Website and download: objectdomain.com
- Commercial, free time-limited demo for <= 30 classes
- Written in Java
- Runs on most platforms
- Needs to export model from its native .odm format

Powerdesigner (by Sybase)

- Website and download: sybase.com
- XMI version 1.1
- Needs to export model

Umbrello (KDE)

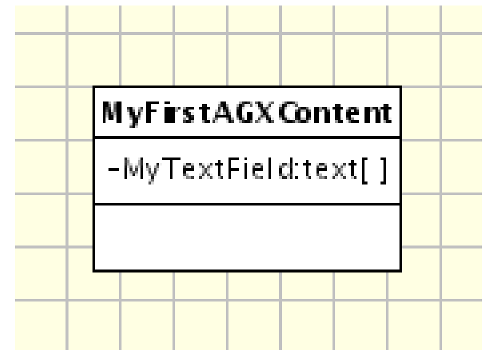
- Website and download: uml.sourceforge.net
- Free software
- Runs under Linux/KDE
- Stores the model natively as XMI
- At the time of testing (somewhere in the first half of 2004), Umbrello wasn't complete and the XMI not 100% standards compliant. Umbrello promises to support XMI correctly on version 1.4, which will be shipped with KDE 3.4.

An almost complete list of UML tools can be found at www.jeckle.de/umltools.htm.

UML

A brief introduction to UML and pointers to further readings.

UML - the Unified Modelling Language - is a graphical language designed to describe software through diagrams. There are several different types of diagrams available, but the ones most relevant to ArchGenXML are:



- The class diagram
- The state diagram

Class diagrams are used to draw interfaces, content types (represented as classes) and tools (represented as classess with the `portal_tool` stereotype), as well as the attributes and public operations on these. In addition, associations in the diagram show how objects are aggregated within or referenced from one another.

The goal of model-driven development is to create the "blueprints" for your software in a well-defined, easily-communicated format: the UML model and diagram thereof. You can design your model using visual tools until you have a structure which adequately represents your needs, and ArchGenXML will generate the necessary code.

You probably have to customise that code somewhat, filling in method bodies, creating new page templates etc., but ArchGenXML takes care of all the boilerplate for you. With tagged values and stereotypes you can customise the generated code with a surprising degree of flexibility and control, and when you need to hand-code something, ArchGenXML won't overwrite your changes (provided you stick to the protected code sections, clearly marked in the source code).

This manual does not aim to teach you UML and object-oriented, model-driven software developement. There are several other fine manuals about that on the web. A very good starting point is the OMG UML Resource Page including its web-links to tutorials.

Getting started

Create your first content type with ArchGenXML

Creating a minimal content type in UML

Open the UML tool of your choice. Make a new UML model and add a class diagram. Choose the tool for class creation and add a class to the diagram. Give it a name such as "MyFirstAGXContent" and add an attribute MyTextField with type `text`. See also: example_1.xmi

Generating the product

Save/export your model as an XMI file with the name MyFirstExample.xmi (or in an XMI-container format like .zargo or .zuml). Then go to the command line and execute:

```
ArchGenXML.py MyFirstAGXExample.xmi
```

ArchGenXML will begin code generation. When it completes, you will have a new folder `MyFirstAGXContent` on your file system. (The folder will be named `MyFirstAGXContent` if that's the name you gave to your model; you can overwrite this output directory with the `-o` option).

Installing and using the generated product

Move the whole folder `MyFirstAGXContent` to your Zope instance's `Products` folder. Restart Zope, open Plone in a browser and log in as manager. Choose `Plone Setup` from the personal bar and choose `Add/Remove Products`. A new product `MyFirstAGXContent` should now appear in the list of products available for install. Choose it and click `install`. Go to your personal folder. In the list of addable items you'll find the new product as an addable content type. Add a test instance to see if it works.

Basics: Classes / Content-Types

How to generate content types, tools and interfaces.

Overview

By default, when you create a class in your class diagram, it represents an Archetypes content type. You can add operations in your model to generate methods on the class, and attributes to generate fields in the schema. The quick reference at the end of this tutorial will tell you which field types you can use. You should also browse the Archetypes quick reference documentation to see what properties are available for each field and widget type. You may set these using tagged values (see below).

There are three basic ways in which you can alter the way your content types are generated:

You may set one or more stereotypes on your class, which alters the "type" of class. A stereotype <<portal_tool>>, for example means you are generating a portal tool rather than just a simple content type.

You may use tagged values in your model to configure many aspects of your classes, their attributes and their methods. A list of recognised tagged values acting on classes, fields and methods are found in the quick reference at the end of this tutorial. When reading tagged values, ArchGenXML will generally treat them as strings, with a few exceptions where only non-string values are permitted, such as the `required` tagged value. If you do not wish your value to be quoted as a string, prefix it with `python:`. For example, if you set the tagged value `default` to `python:["high", "low"]` on a `lines` attribute, you will get `default=["high", "low"]` in a LinesField in your schema.

ArchGenXML is clever about aggregation and composition. If your class aggregates other classes, it will be automatically made into a folder with those classes as the allowed content types. If you use composition (signified by a filled diamond in the diagram) rather than aggregation, the contained class will only be addable inside the container, otherwise it will be addable globally in your portal by default.

Variants of Content Types

Simple Classes

A simple class is what we had in MyFirstAGXContent in the previous chapter. A simple class is based on `BaseContent`. This is the default if no other options override.

Folderish Classes

The easiest way to make a content type folderish is to introduce composition or aggregation in your model - the parent class will become folderish and will be permitted to hold objects of the child classes. You can also make a class folderish just by giving it the <<folder>> stereotype. Both of these approaches will result in an object derived from `BaseFolder`.

In the most recent version of ArchGenXML you can also give a class the <<ordered>> stereotype (possibly in addition to <<folder>>) in order to make it derive from `OrderedBaseFolder` and thus have ordering support. Alternatively, you can set the `base_class` tagged value on the class to `OrderedBaseFolder`. This is a general technique which you can use to

override the base folder should you need to. As an aside, the `additional_parents` tagged value permits you to derive from multiple parents.

Other tagged values which may be useful when generating folders are:

filter_content_types
: Set this to `0` or `1` to turn on/off filtering of content types. If content types are not filtered, the class will act as a general folder for all globally addable content.

allowed_content_types
: To explicitly set the allowable content types, for example to only allow images and documents, set this to: `Image, Document`. Note that if you use aggregation or composition to create folderish types as described above, setting the allowed content types manually is not necessary.

Portal tools

A portal tool is a unique singleton which other objects may find via `getToolByName` and utilise. There are many tools which ship with Plone, such as portal_actions or portal_skins. To create a portal tool instead of a regular content type, give your class the `<<portal_tool>>` stereotype. Tools can hold attributes and provide methods just like a regular content type. Typically, these hold configuration data and utility methods for the rest of your product to use. Tools may also have configlets - configuration pages in the Plone control panel. See the quick reference at the end of this document for details on the tagged values you must set to generate configlets.

CMFMember memberdata objects

In the most recent version of ArchGenXML, you can use the `<<member>>` stereotype to make your class derive from CMFMember's Member base object and get the standard CMFMember memberdata fields. Upon installation, your custom member type will be registered with CMFMember.

Abstract mixin classes

By marking your class as `abstract` in your model (usually a separate tick-box), you are signifying that it will not be added as a content type. Such classes are useful as mixin parents and as abstract base classes for more complex content types, and will not have the standard Archetypes registration machinery, factory type information or derive from BaseClass.

Stub classes

By giving your class the `<<stub>>` stereotype, you can prevent it from being generated at all. This is useful if you wish to show content types which are logically part of your model, but which do not belong to your product. For instance, you could create a stub for Plone's standard Image type if you wish to include this as an aggregated object inside your content type - that is, your content type will become folderish, with Image as an allowable contained type.

Deriving/Subclassing Classes

Deriving or subclassing a class is used to extend existing classes, or change their behavior. Using generalisation arrows in your model, you can inherit the methods and schema from another content type or mixin class in your class.

Simple Derivation

All content types in Archetypes are derived from one of the base classes - BaseContent, BaseFolder, OrderedBaseFolder and so on. If you wish to turn this off, for example because the base class is being inherited from a parent class, you can set the `base_class` tagged value to `0`.

Multiple Derivation

You can of course use multiple inheritance via multiple generalisation arrows in your model. However, if you need to use a base class that is not on your model, you can set the `additional_parents` tagged value on your class to a comma-separated list of parent classes.

Deriving from other Products

If you want to derive from a class of an other product create a stub class with a tagged value 'import_from': This will generate a import line `from VALUE import CLASSNAME` in classes derived from this class.

Interfaces

Interfaces are a way of formally documenting the public interface to your code. By convention, they are usually in the `interfaces` package (see below). Use your UML modeller's interface tool to create new interfaces.

Interfaces do not have most of the added fluff that content types do - they do not even have method bodies. They do, however, have extensive documentation. A class is said to "realise" an interface when it provides implementations for the methods defined in the interface. The UML realisation arrow (a dotted line with an empty arrowhead) will ensure that your content types are linked to the correct interfaces by way of the `__implements__` class attribute.

Packages - bring order to your code

Packages are both a UML concept and a Python concept. In Python, packages are directories under your product containing a set of modules (.py files). In

UML, a package is a logical grouping of classes, drawn as a large “folder” with classes inside it. To modularise complex products, you should use packages to group clases together.

Basics: Attributes / Fields

How to control the fields of your schema.

The schema of your content types, generated from the attributes of your model and their tagged values, contains Archetypes fields. Each field has a type and a widget. The Archetypes documentation and the quick reference at the end of this document describes which fields are available and what parameters they take as configuration.

usage of tagged values

If you set a tagged value on an attribute of your class, in general that tagged value will be passed through as a parameter to the generated Archetypes field. Hence, if you set a tagged value `enforceVocabulary` to the value `1` on an attribute, you will get `enforceVocabulary=1` for that field in the generated schema. Similarly, you can set a field's widget properties by prefixing the tagged value with `widget:`. `widget:label` sets the label of a widget, for instance.

non-string tagged values

As before, when reading tagged values, ArchGenXML will generally treat them as strings, with a few exceptions where only non-string values are permitted, such as the `required` tagged value. If you do not wish your value to be quoted as a string, prefix it with `python:`. For example, if you set the tagged value `default` to `python:["high", "low"]` on a `lines` attribute, you will get `default=["high", "low"]` in a LinesField in your schema.

index in catalog

To create an index in portal_catalog for this field add the tagged value `index` with value `FieldIndex`. An FieldIndex with the name of the fields accessor (e.g. get<Fieldname>) gets created.

Multiple indexes can be defined in a tuple, indexes for special catalogs can be prefixed with the catalog name following a `/` (e.g. `python:("FieldIndex", "member_catalog/TextIndex")`)

To include the index in catalog metadata (and have the attribute ready to use in the brain objects), append `:brains` (same as older `:schema`), (e.g. `FieldIndex:brains`)

Basics: Widgets

Simple content type creation: Setting up the Widgets.

ArchGenXML will pick a default widget for your fields and fill in default labels and descriptions. For example, a `string` field gets a `StringWidget` by default. You can override this in two ways.

First of all, you can set a tagged value `widget` on your field and provide the code for the entire widget definition. This method is depreciated

in favour of individual widget properties, which make it much easier to manage your widgets, however.

Widget options are specified with the prefix `widget:`. As with normal field tagged values, unrecognised options will be passed straight through to the widget definition.

The most common widget options are:

widget:label
: sets the widget's label

widget:description
: sets the widget's description

widget:label_msgid
: overrides the default label message id (i18n)

widget:description_msgid
: overrides the default description message id (i18n)

widget:i18n_domain
: sets the i18n domain (defaults to the product name)

You may also use widget-specific options, such as `widget:size` where they apply.

Changing the widget of a field

Let's assume you use a StringField for capturing the type of a fruit and you know that you'll just have 5 types of fruit to select from (apple, peach, pear, banana and cherry). It's probably the best to use a `SelectionWidget`, coupled with a vocabulary set on the field (by setting the tagged value `vocabulary` to `python:["apple", "peach", "pear", "banana"]`) to restrict the user to these addable types.

The first way to achieve this may be to use the `widget` tagged value to set the entire widget in one go. For example, you could write:

```
SelectionWidget(
label="""Fruit type""",
description="""Select one of the fruits""",
label_msgid='label_fruit_type',
description_msgid='help_fruit_type',
i18n_domain='fruit',
),
```

However, that method is depreciated in the latest version of ArchGenXML, where you can set the property `widget:type` to be the name of your chosen widget type, such as `StringWidget` or `SelectionWidget`. In previous versions of ArchGenXML, you can accomplish the same thing by setting `widget` to be the name of your widget only, and use the spe-

cific tagged values (such as `widget:label`) to set the fields of the widget explicitly.

Using custom widgets

You have two options to change the type of the widget to a custom type, a type outside Archetypes base framework:

To change the type for one field use `widget:type` and set it to `MyCustomWidget` if you want to use `MyCustomWidget`

To change a the widget used for one field-type for the whole model, a product, a package or just for all fields in one class you can set on mode, product, package or class level the tagged value `default:widget:FIELDNAMEABBREVIATION` to `WIDGETNAME`. For example use the tagged value `default:widget:Reference` and set it to `ReferenceBrowserWidget` to use the ReferenceBrowserWidget instead of the ReferenceWidget. You might also want to use also the `additional_imports` tagged value and set it to `from ATReferenceBrowserWidget.ATReferenceBrowserWidget import ReferenceBrowserWidget` on your class to ensure that you get the widget definition imported into your class.

Basics: Methods and Actions

Simple Content-Type creation: Defining Methods and Actions

To create a method in your class, add a method to the UML diagram, with the desired parameters. The types of the parameters and the type of the return value are ignored, since Python does not support this.

Methods can different access specifiers (also called visibilities) These are:

public (shown by a + before the method name)

The method is part of the class' public interface. It will be declared public (accessible from unsafe/through-the-web code) by default. If you add a tagged value `permission` (see below), it will be declared as protected by this permission.

protected (#)

The method is not part of the class' public interface, but is meant for use by subclasses. It will be declared private to prevent unsafe code from accessing it.

private (-)

The method is internal to the class. It will be declared private to prevent unsafe code from accessing it.

package (~)
: The method is intended to be accessed by other code in the same module as the class. It will not gain any Zope security assertions, relying instead on the class/module defaults.

There are a few tagged values you can use to alter how the code is generated:

code
: Sets the python code body of the method. Only use this for short one-liners. If you fill in code manually in the generated files, method bodies will be preserved when you re-generate the product from the UML model.

documentation
: Content of the python doc-string of the method. You can also use the documentation feature of most UML modellers to set documentation strings.

permission
: Applies to methods with `public` visiblity only. If you set the permission tagged value to `My custom permission` results in security.declareProtected("""My custom permissi on""",`methodname`) - that is, access to your method is protected by the permission with the name `My custom permission`.

If you want to use the CMF core permissions, add an `imports` tagged value to the method's class containing `from Products.CMFCore import CMFCorePermissions`, and then set the permission tagged value of your method to `python:CMFCorePermissions.View`, `python:CMFCorePermissions.ModifyPortalContent` or any other core permission. You can also use the common paradigm of defining permissions in config.py as constants with names like EDIT_PERMISSION. With newer versions of ArchGenXML, a config.py is automatically generated and its contents imported, so you can just set the permission tagged value to, for example, `python:EDIT_PERMISSION`.

Archetypes uses actions and forms for generating custom tabs for accessing data of an Archetype object. ArchGenXML can generate actions and forms for you: Just define a method without any parameters and set its stereotype to one of the following three:

<<action>>
Generates a general action.

<<view>>
Generates an action and copies an empty page template to the skins directory, with the name `<methodname>.pt`. If there is already such a template, it will be left untouched.

<<form>>
Generates an action and copies an empty form-controller template to the skins directory. Its name is generated by using `<methodname>.pt`. If there is already such a file, it will be left untouched.

Once again tagged values can be set on the sterotyped methods in order to set some properties of the action:

action
The TAL expression representing the action to be executed when the user invokes the action. Defaults to the methodname.

category
The category of an action, view or form. Defaults to `object`.

id
The id of an action, view or form. Defaults to the methodname.

label
The label of an action, view or form. Defaults to the methodname.

permission
`permission=My permission` results in 'permissions': ('"My Permission"',). See the description of the general `permission` tagged value above for more.

condition
A TALES expression giving a condition to control when the action is to be made available.

form
see action.

view
see action.

You can override the default Archetypes actions by using special names for the id. These are:

view
: for overriding the default view action.

edit
: for overriding the default edit action.

contents
: for overriding the default contents action.

Basics: Relationships between classes and objects

How to use references, associations, aggregations and compositions

With aggregations, compositions and associations you define where your new type will show up, what it might contain and to which content it can point to.

There is virtually no limit on how many aggregations, compositions and associations you can attach to a class.

Aggregations: Global Containment

Aggregation means: This content can exist global and in this container. The container class that gets the empty rhomb attached is derived from BaseFolder and it's allowed_content_types is set to the class that is attached to it.If more than class is attached to one class by aggregations the allowed_content_types is extended accordingly. The attached class keeps the default **global_allow=1**.

Compositions: Strict Containment

Compositions are used to model parts that exist or live and die with their associated owner. So the code generated is similair to the one generated by aggregations, but with one major difference: The attached classes are only allowed to be generated in the folderish type of the class they're attached to (this is done by setting **global_allow=0** in the factory type information of the class).

Directed Associations: References

References are used to store the relation of an object to other objects.

Each content type that derives from **IReferenceable** is capable of beeing referenced. Objects from such a content type have an UID (Unique Identification) that's unique throughout the whole Plone site. Therefore References don't break if you move referenced objects around in the site.

To use **ReferenceFields** there are two possible ways. The by models-design clean way is to use directed associations. Another possibility is to define References as class-attributes.

Directed Associations

An directed association between two classes generates a **Reference-Field** in the class where the association starts.

The **relationship** itself is named after the association's name.

The multiplicity defines if the allows a 1:1 or 1:n relation. Attention: This only results in validation on the field. References at all don't know anything about multiplicity, so this is only a check on userinterface-level.

All other field settings are taken from the association's end, including information how to generate the widget. By default a ReferenceWidget is used. You can use tagged values on the association's end to define label, description, a different widget-type, schemata, etc. like you do it on a field (on a class attribute).

The big drawback of using associations to create ReferenceFields is that they always get attached to the end of the schema and there is no way to change that in the UML diagram. So if you need order in your fields read the next section.

References as class attributes

You can define an attribute with the type reference. Then you can apply any needed tagged values to it.

keys of interest are:

```
  allowed_types : needs a list of allowed types
  multiValued   : set to 0 to only be able to select one
object to reference to
  relationship  : name of the relationship in the
reference_catalog
```

The benefit of using an attribute to define the reference is that you can define the place in the schema where the ReferenceField will show up.

Reference classes (advanced)

Sometimes it's needed to store information not in the origin or destination class, but in the reference itself. UML has a notation to model this: association classes.

ArchGenXML support them automatically. When a model includes an association class, two things occur:

A new content type is created, named like the association name

The generated ReferenceField has a new attribute defined like this:

referenceClass = ContentReferenceCreator('My_Association_Name')

This causes that the class of the reference instances is now not "`Arche-`

types.ReferenceEngine.Reference", but "Archetypes.ReferenceEngine.ContentReference", a subclass of it that has a new method: getContentObject(), that return the content inside the reference.

The same effect can be reached without association classes, by defining a content type and then adding the "association_class" tagged value to the association (although I haven't been able to make this work).

To create the reference via code, use a special form of the addReference method:

```
origin = <the origin content>
destination = <the destination content>
assocName = <the association name>
origin.addReference(destination,
assocName,
referenceClass=ContentReferenceCreator(assocName),
attr1=value1,
attr2=value2...)
(where attr1, attr2... are the attributes of the
association)
```

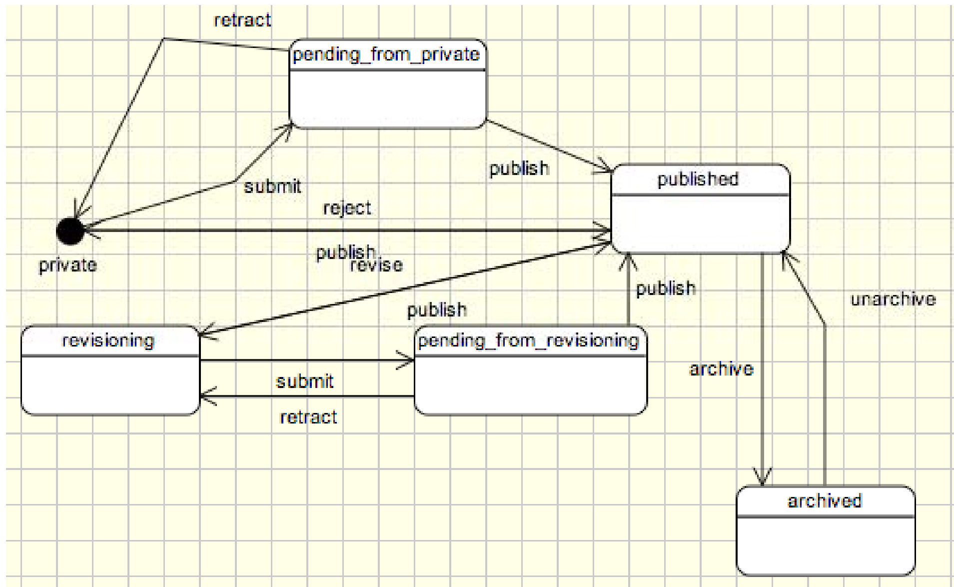
To read the data, we can't use the origin.getRefs(assocName) method, as usual, because it returns only the destination objects. One way to read it is by using the reference_catalog tool:

```
from Products.CMFCore.utils import getToolByName
tool = getToolByName(origin, 'reference_catalog')
refs = tool.getReferences(origin, assocName)
if not refs:
return []
else:
return [(ref.getContentObject(), ref.getTargetObject())
for ref in refs]
```

Basics: Workflow Generation

ArchGenXML can use state diagrams to create custom workflows for your types.

ArchGenXML can use state diagrams to generate workflows for a portal type. Workflows are used to set the various states an object can be in, and the transitions between them.



Importantly, workflows control permissions of objects. By convention, and for convenience and consistency, most content types will use the permissions found in the `CMFCorePermissions` class in the CMFCore product to control access to their methods. The methods generated by and inherited from the CMF and Archetypes frameworks adhere to this principle. Although many different content types use the same basic permissions to control access, workflows are the means by which you can control permissions for an object in detail. For instance, you may wish to specify that in the `testing` state, Manager and Reviewer has `Modify portal content` permissions, and Owner, Manager and Reviewer has `View` permissions. For the `completed` state, you could have a different set of permissions. See the DCWorkflow documentation for more details about how to use workflows.

Problems with UML-Software

The workflow implementation of ArchGenXML has to date only been tested with ArgoUML and Poseidon (tested Version is 3.2 CE).

ObjectDomain is known not to work at this time, because it does not appear to correctly export the XMI for state diagrams. If you have different experiences, please add a comment to this document or contact us.

Creating a workflow

In your UML modeller, add a state diagram for the class you wish to create a custom workflow for. If you don't want to assign the workflow to a class use an class with stereotype `stub`. In Poseidon, this is done by right-clicking on the object in the tree on the left hand side, and selecting to add a new state diagram. The name of the state diagram becomes the name of the workflow.

States

On the state diagram, add a state item (a rounded-corner box) for each state. You must have an initial state of your workflow for it to work correctly. Use a "initial state" symbol (filled cirlce) for the state your object defaults to after creation. Optional you can use a normal state item and set a tagged value `initial_state` with value 1 to it.

At present, ArchGenXML does not support the "final state" UML symbols to represent final states, so you should stick to the standard state symbols.

The names of your states in UML become the names of the states in your workflow. The user-visible label can be set with the `label` tagged value; it defaults to the state name.

Transitions

For each possible transition between states, add a transition arrow to your UML model. The name of the transition becomes the name of the workflow action. You can set the `label` tagged value on the transition to set a custom label to display to the user.

Transition guards

You can add a guard to a transition to restrict to whom and when it is made available. Set the `expression` field of a transition to a |-separated list of the following pairs:

guard_roles
: Set `guard_roles:Owner; Manager` to restrict the transition to users posessing the Owner or Manager role in the current context.

guard_permissions
: Set `guard_permissions:My custom permission;View` to ensure that only those users with `My custom permission` or `View` permissions in the current context are allowed to access the transition.

guard_expr
: Set 'guard_expr:expression", where `expression` is a TALES expression, to have the expression be evaluated in order to determine whether the transition should be made available.

Thus, to restrict access to roles Reviewer and Manager, and only those users with permission `My custom permission` and `View` in the current context, you can set the expression of the transition to `guard_roles: Reviewer;Manager|guard_permissions:My custom permission, View`.

If you are using Poseidon, transition guards are located in the property of the transition arrow with the name `[A] Guard`. You can add an expression like the one outlined above to this field.

Permissions

ArchGenXML uses tagged values on states in a somewhat unconventional, though convenient, way to control permissions. With the exception of the special-case `initial_state` and `label` tagged values, you give the name of the permission as the tagged value key, and a comma-separated list of roles the permission should be enabled for as the value.

There are three shorthand permission names available:

access
: referes to the `Access contents information` permission.

view
: refers to the `View` permission.

modify
: refers to the `Modify portal content` permission.

Hence, if you want your state to permit anonymous users and members to view your content, only permit managers to modify, and permit both the owner and managers to add new objects controlled by the `Add MySubTypes` permission, you can add three tagged values to the workflow state:

```
view           ==> Anonymous, Member
modify         ==> Manager
Add MySubTypes ==> Owner, Manager
```

Workflow actions

The `portal_workflow` tool allows a script to be executed before and/or after a transition is completed. ArchGenXML lets you specify the names of these actions, and will generate an external method for you to implement for each uniquely named action in your workflow.

Actions are set using the `effect` field of a transition. The value given here gives the name of the action method to execute (and thus must be valid python method name). ArchGenXML will create or modify a script containing external methods for each workflow, in `Extensions/<WorkflowName>_scripts.py` in your product. You must fill in the method bodies for the actions in this script. Method bodies will be preserved upon re-generation of your product from the UML model.

By default, actions specified in this way are post-transition actions, meaning that they are executed **after** the transition has taken place. If you wish to specify a pre-transition action, executed before the transition takes place, separate action names by semicolons: `preActionName;postAc`

tionName. If you want only a pre-transition action, use preActionName; to specify that there is an empty post-transition action.

Attach workflow to more than one class

In UML there is no semantic to use a workflow for more than one class. We introduced the tagged value use_workflow for classes. Value is the workflow name.

Worklist support

You can attach objects in a certain state to a worklist. A worklist is something like the "documents to review" list you get when you're a reviewer in a Plone site. This is done by adding a tag worklist to the state with the name of the worklist as value (like review_list).

You can add more than one state to a worklist, just by specifying the same name for the worklist tagged value. Likewise, you can have more than one worklist (just not on the same state). The tagged value worklist:guard_permissions allows you to specify the permission you need to have to view the worklist. The default value is Review portal content.

Third Party Product Integration: ATVocabularyManager

ATVocabularyManager is a product for letting site managers define vocabularies for fields through-the-web or by import from XML files.
ArchGenXML can generate the necessary code to use this product.

ATVM manages dynamic vocabularies. It installs a tool, where a site Manager can add, change and delete vocabularies. These vocabularies can then be used anywhere on the site.

You can download ATVocabularyManager from the archetypes subversion repository: http://svn.plone.org/archetypes/ATVocabularyManager/

Adding ATVM-vocabs to your UML model is quite easy.

Add a selection or multiselection field to your type.

Add a tag vocabulary:name and give it a name, let's say countries

Add a tag vocabulary:type with the value ATVocabularyManager

We are now finished with the UML. Save it and let AGX do the work. What still is missing, is to install the countries vocabulary. Therefore:

Add a file called AppInstall.py in the /Extensions folder of your product

Add the following code:

```
from Products.ATVocabularyManager.config import TOOL_NAME as
ATVOCABULARYTOOL
```

```
from Products.CMFCore.utils import getToolByName
def install(self):
"""
let's install the countries vocab
"""
vocabs = {}
vocabs['countries'] = (
('ice', u'Iceland'),
('nor', u'Norway'),
('fin', u'Finland'),
('tyr', u'Tyrol'),
('auf', u'Ausserfern'),
)
portal=getToolByName(self,'portal_url').getPortalObject()
atvm = getToolByName(portal, ATVOCABULARYTOOL)
for vkey in vocabs.keys():
# create vocabulary if it doesnt exist:
vocabname = vkey
if not hasattr(atvm, vocabname):
# print >>out, "adding vocabulary %s" % vocabname
atvm.invokeFactory('SimpleVocabulary', vocabname)
vocab = atvm[vocabname]
for (ikey, value) in vocabs [vkey]:
if not hasattr(vocab, ikey):
vocab.invokeFactory('SimpleVocabularyTerm', ikey)
vocab[ikey].setTitle(value)
```

This sets up a vocabulary `countries` with the given values, and registers it with ATVocabularyManager.

Third Party Product Integration: Relations

Create relations between portal-types model-driven. Support for Relations Product (complex references). Define sets of rules for validation, creation and lifetime of Archetypes references. ArchGenXML can generate the necessary code and XML-configuration data to use this product.

Prerequisites

To enable Relations install the Product (code-location).

Basics

As an option on command line, up to a tagged-value on model-level or on a single UML-Assoziation you just define the `relation_implementation` and set it to `relations`. A directed Assoziation results in one Relation.

Give the assoziation and its assoziation ends names. They'll be used as the names for the RelationField.

Inverse Relation

If the assoziation is not directed (navigable on both assoziation ends) an inverse relation will be created.

The tagged-value `inverse_relation_name` will be used for the backrelation on undirected assoziations. It defaults to the assoziation name postfixed by `_inverse`.

Cardinality

You can use the Multiplicity on in UML to define the cardinality of an Relation.

You can use the minimum and maximun value here using the syntax `1..5` which means at least one relation but not more than five.

Constraints

type-constraint
: as described above an assoziation between two portal-types will be created.

interface-constraint
: an assoziation between an archetypes class and an interface will create an interface-constraint. the relation is allowed to all classes implementing this interface.

Association classes

Association classes can be used to store data on the relation as an object. You can model it using the UML association class or using a tagged value `association_class` on the association.

Quick Reference

A very nice Quick Reference for ArchGenXML abusers.

Complete list of the field types including their default settings:

string

StringField

StringField

searchable=1

text

TextField

StringField

searchable=1

TextAreaWidget()

richtext

TextField

TextField

default_output_type=text/html

allowed_content_types=(text/plain,text/structured,text/html,application/msword,)

selection

StringField with SelectionWidget

StringField

multiselection

LinesField with SelctionWidget

LinesField

multiValued=1

integer

IntegerField

IntegerField

searchable=1

float

Floatfield

FloatField

searchable=1

DecimalWidget()

boolean

BoleanField

BooleanField

searchable=1

lines

LinesField

LinesField

searchable=1

date

DateTimeField

DateTimeField

searchable=1

image

ImageField

ImageField

sizes ={'small':(100,100),'medium':(200,200),'large':(600,600)}

AttributeStorage()

file

FileField

FileField

AttributeStorage()

FileWidget()

lines

LinesField

LinesField

searchable=1

fixedpoint

FixedPointField

FixedPointField

date

DateField

DateTimeField

AttributeStorage()

reference

ReferenceField

ReferenceField

backreference

BackReferenceField

BackReferenceField

computed

ComputedField

ComputedField

photo

PhotoField

PhotoField

Tagged values for fields:

searchable

register and index the field in the catalog,

1 .. register and index

0 .. don't register and index

storage

AttributeStorage(), SQLStorage(), ...

sizes

defines the sizes of the images in a ImageField

example: python:{'small':(80,80),'med ium' :(200,2000),'large':(600,600)}

default_method

no idea what that does

required

defines whether a field should be rendered required, or not.

1 .. field is required

0 .. field is not required

accessor

defines the accessor of a field

vocabulary

defines the vocabulary or the method generating a vocabulary

allowed_types

defines the allowed types in a ReferenceField

relationship

defines the relationship, used in a ReferenceField

multiValued

defines whether a SelectionField accepts one or more values,

1 .. multivalued

0 .. singlevalued

These tagged values are just the ones handy for fields, the full lists of tagged values and stereotypes are shown on the next two pages.

Tagged value overview

This pages provides the full overview of the tagged values that are recognised by ArchGenXML.

action/form/view

action
: For a stereotype `action`, this tagged value can be used to overwrite the default URL (`..../name_of_method`) into `..../tagged_value`.

action_label
: TODO.

category
: The category for the action. Defaults to `object`.

condition
: A TALES expresson defining a condition which will be evaluated to determine whether the action should be displayed.

documentation
: You can add documention via this tag; it's better to use your UML tool's documentation field.

form
: For a stereotype `form`, this tagged value can be used to overwrite the default URL (`..../name_of_method`) into `..../tagged_value`.

id
: The id of the action. Use `id`,

label
: The label of the action - displayed to the user.

view
: For a stereotype `view`, this tagged value can be used to overwrite the default URL (`..../name_of_method`) into `..../tagged_value`.

association

association_class
: You can use associations classes to store content on the association itself. The class used is specified by this setting. Don't forget to import the used class properly.

association_vocabulary
: Switch, defaults to False. Needs Product `ATVocabularyManager`. Generates an empty vocabulary with the name of the relation.

back_reference_field
: Use a custom field instead of ReferenceField.

documentation
: You can add documention via this tag; it's better to use your UML tool's documentation field.

inverse_relation_name
: Together with `Relations` Product you have inverse relations. the name default to `name_of_your_relation_inverse`, but you can overrrule it using this tagged value.

label
: Sets the readable name.

reference_field
: Use a custom field instead of ReferenceField.

relation_field
: Use a custom field instead of RelationField. Works only together with `Relations` Product and relation_implementation set to `relations`.

relation_implementation
: Sets the type of implementation is used for an association: `basic` (used as default) for classic style archetypes references or `relations` for use of the `Relations` Product.

attribute

accessor
: Set the name of the accessor (getter) method. If you are overriding one of the DC metadata fields such as `title` or `description` be sure to set the correct accessor names such as `Title` and 'Description'; by default these accessors would be generated as getTitle() or getDescription().

copy_from
: To copy an attribute from another schema, give it the type `copy`. The tagged value `copy_from` is then used to specify which schema to copy it from (for instance, `BaseSchema` when copying Description from the base schema). For copying your own schemas, add an `imports` tagged value to import your class (say `MyClass`) and then put `MyClass.schema` in your `copy_from` value.

default
: Set a value to use as the default value of the field.

default_method
: Set the name of a method on the object which will be called to determine the default value of the field.

documentation
: You can add documention via this tag; it's better to use your UML tool's documentation field.

enforceVocabulary
: Set to true (1) to ensure that only items from the vocabulary are permitted.

label
: Sets the readable name.

multiValued
: Certain fields, such as reference fields, can optionally accept morethan one value if multiValued is set to true (1)

mutator
: Similarly, set the name of the mutator (setter) method.

original_size
: Sets the maximum size for the original for an ImageField widget.

read_permission
: Defines archetypes fields read-permission. Use it together with workflow to control ability to view fields based on roles/permissions.

required
: Set to true (1) to make the field required

schemata
: If you want to split your form with many, many attibutes in multiple schemata ("sub-forms"), add a tagged value `schemata` to the attributes you want in a different schemata with the name of that schemata (for instance "personal data"). The default schemata is called "default", btw.

searchable
: Whether or not the field should be searchable when performing a search in the portal.

sizes
: Sets the allowed sizes for an ImageField widget.

source_name
: With attribute type `copy` sometimes schema-recycling is fun, together with copy_from you can specify the source name of the field in the schema given by copy_from.

validation_expression
: Use an ExpressionValidator and sets the by value given expression.

validators
: TODO.

vocabulary
: Set to a python list, a DisplayList or a method name (quoted) which provides the vocabulary for a selection widget.

vocabulary:name
: Togther with Products `ATVocabularyManager` this

sets the name of the vocabulary.

vocabulary:type
Enables support for Products `ATVocabularyManager` by setting value to `ATVocabularyManager`.

widget
Allows you to set the widget to be used for this attribute.

widget:description
Set the widget's description

widget:description_msgid
Set the description i18n message id. Defaults to a name generated from the field name.

widget:i18n_domain
Set the i18n domain. Defaults to the product name.

widget:label
Set the widget's label

widget:label_msgid
Set the label i18n message id. Defaults to a name generated from the field name.

widget:type
Set the name of the widget to use. Each field has an associated default widget, but if you need a different one (e.g. a SelectionWidget for a string field), use this value to override.

write_permission
Defines archetypes fields write-permission. Use it together with workflow to control ability to write data to a field based on roles/permissions.

class

additional_parents
A comma-separated list of the names of classes which should be used as additional parents to this class, in addition to the Archetypes BaseContent, BaseFolder or OrderedBaseFolder. Usually used in conjunction with `imports` to import the class before it is referenced.

allow_discussion
Whether or not the content type should be discussable in the portal by default.

allowed_content_types
A comma-separated list of allowed sub-types for a (folderish) content type. Note that allowed content types are automatically set when using aggregation and composition between classes to specify containment.

archetype_name
The name which will be shown in the "add new item" drop-down and other user-interface elements. Defaults to the class

name, but whilst the class name must be valid and unique python identifier, the archetype_name can be any string.

author

You can set the author project-wide with the `--author` commandline parameter (or in the config file). This tag allows you to overwrite it on a class level.

base_actions

Sets the base actions in the class's factory type information (FTI).

base_class

Explicitly set the base class of a content type, overriding the automatic selection of BaseContent, BaseFolder or OrderedBaseFolder as well as any parent classes in the model. See also additional_parents.

base_schema

Explicitly set the base schema for a content type, overriding the automatic selection of the parent's schema or BaseSchema, BaseFolderSchema or OrderedBaseFolderSchema.

class_header

An arbitrary string which is injected into the header section of the class, before any methods are defined.

contact_schema

TODO. CMFMember related.

content_icon

The name of an image file, which must be found in the skins directory of the product. This will be used to represent the content type in the user interface.

copyright

You can set the copyright project-wide with the `--copyright` commandline parameter (or in the config file). This tag allows you to overwrite it on a class level.

creation_permission

Sets the creation permission for the class. Example: `Add portal content`.

creation_roles

You can set an own role who should be able to add a type. Use an Tuple of Strings. Default and example for this value: `("Manager", "Owner", "Member")`.

default_actions

If set to true (1), generate explicitly the default `view` and `edit` actions. Usually, these are inherited from the Archetypes base classes, but if you have a funny base class, this may be necessary.

default_view

The TemplateMixin class in Archetypes allows your class to present several alternative view templates for a content type. The default_view value sets the default one. Defaults to `base_view`. Only relevant if you use TemplateMixin.

disable_polymorphing

Normally, archgenxml looks at the parents of the current class for content types that are allowed as items in a folderish class. So: parent's allowed content is also allowed in the child. Likewise, subclasses of classes allowed as content are also allowed on this class. Classic polymorphing. In case this isn't desired, set the tagged value `disable_polymorphing` to 1.

doctest_name

In a tests package, setting the stereotype `<<doc_testcase>>` on a class turns it into a doctest. The doctest itself is placed in the doc/ subdirectory. The `doctest_name` tagged value overwrites the default name for the file (which is the name of the doctestcase class + `.txt`). ArchGenXML appends the `.txt` extension automatically, so you don't need to specify it.

documentation

You can add documention via this tag; it's better to use your UML tool's documentation field.

email

You can set the email project-wide with the `--email` commandline parameter (or in the config file). This tag allows you to overwrite it on a class level.

filter_content_types

If set to true (1), explicitly turn on the filter_content_types factory type information value. If this is off, all globally addable content types will be addable inside a (folderish) type; if it is on, only those values in the allowed_content_types list will be enabled. Note that when aggregation or composition is used to define containment, filtered_content_types will be automatically turned on.

folder_base_class

Useful when using the `<<folder>>` stereotype in order to set the folderish base class.

folderish

Explicitly specify that a class is folderish. It is usually better to the the `<<folder>>` stereotype instead.

global_allow

Overwrite the AGX-calculated `global_allow` setting. Setting it to `1` makes your content type addable everywhere (in principle), setting it to `0` limits it to places where it's explicitly allowed as content.

hide_actions
: A comma- or newline-separated list of action ids to hide on the class. For example, set to `metadata, sharing` to turn off the metadata (properties) and sharing tabs.

hide_folder_tabs
: Hides the folder tabs for this content type. (Mostly the “Contents” tab).

immediate_view
: Set the immediate_view factory type information value. This should be the name of a page template, and defaults to `base_view`. Note that Plone at this time does not make use of immediate_view, which in CMF core allows you to specify a different template to be used when an object is first created from when it is subsequently accessed.

import_from
: If you wish to include a class in your model (as a base class or aggregated class, for example) which is actually defined in another product, add the class to your model and set the import_from tagged value to the class that should be imported in its place. You probably don’t want the class to be generated, so add a stereotype `<<stub>>` as well.

imports
: A list of python import statements which will be placed at the top of the generated file. Use this to make new field and widget types available, for example. Note that in the generated code you will be able to enter additional import statements in a preserved code section near the top of the file. Prefer using the imports tagged value when it imports something that is directly used by another element in your model. You can have several import statements, one per line, or by adding several tagged values with the name `imports`.

inherit_allowed_types
: By default, a child type will inherit the allowable content types from its parents. Set this property to false (0) to turn this off.

label
: Sets the readable name.

license
: You can set the license project-wide with the `--license` commandline parameter (or in the config file). This tag allows you to overwrite it on a class level.

login_info_schema
: TODO. CMFMember related.

marshall
: Specify a marshaller to use for the class’ schema.

marshaller
: Specify a marshaller to use for the class’ schema.

migrate_dynamic_view_fti
Migrates FTI of a type/class to CMFDynamicViewFTI. This works only if the class derives from an ATContentType, from ATCTMixIn or direct from CMFDynamicViewFTI. browserdefault.BrowserDefaultMixin.

module
Like `module_name`, it overwrites the name of the directory it'd be normally placed in.

module_name
Like `module`, it overwrites the name of the directory it'd be normally placed in.

plone_schema
TODO. CMFMember related.

policy
TODO.

portal_type
Sets the CMF portal-type this class will be registered with, defaults to the class-name.

read_permission
Defines archetypes fields read-permission. Use it together with workflow to control ability to view fields based on roles/permissions.

rename_after_creation
Setting this boolean value enables or disables explicit the after creation rename feature using `_at_rename_after_creation` class-attribute.

security_schema
TODO. CMFMember related.

suppl_views
The TemplateMixin class in Archetypes allows your class to present several alternative view templates for a content type. The suppl_views value sets the available views. Example: `("my_view", "myother_view")`. Defaults to `()`. Only relevant if you use TemplateMixin.

typeDescription
A description of the type, a sentence or two in length. Used to describe the type to the user.

use_portal_factory
Setting this boolean value enables the registration of the type for use with portal_factory.

use_workflow
Tie the class to the named workflow. A state diagram (=workflow) attached to a class in the UML diagram is automatically used as that class's workflow; this tagged value allows you to tie the workflow to other classes.

write_permission
Defines archetypes fields write-permission. Use it together with workflow to control ability to write data to a field based on roles/permissions.

field

description
Sets an description for this field. It's used for field documentation while registering inside Archetypes.

documentation

You can add documention via this tag; it's better to use your UML tool's documentation field.

label

Sets the readable name.

validation_expression

Use an ExpressionValidator and sets the by value given expression.

method

code

The actual python code of the method. Only use this for simple one-liners. Code filled into the generated file will be preserved when the model is re-generated.

documentation

You can add documention via this tag; it's better to use your UML tool's documentation field.

label

Sets the readable name.

permission

For method with public visibility only, if a permission is set, declare the method to be protected by this permission. Methods with private or protected visiblity are always declared private since they are not intended for through-the-web unsafe code to access. Methods with package visibility use the class default security and do not get security declarations at all.

model

association_class

You can use associations classes to store content on the association itself. The class used is specified by this setting. Don't forget to import the used class properly.

association_vocabulary

Switch, defaults to False. Needs Product `ATVocabularyManager`. Generates an empty vocabulary with the name of the relation.

author

You can set the author project-wide with the `--author` commandline parameter (or in the config file). This tag allows you to overwrite it on a model level.

copyright

You can set the copyright project-wide with the `--copyright` commandline parameter (or in the config file). This tag allows you to overwrite it on a model level.

creation_permission

Sets the creation permission for the class. Example: `Add portal content`.

creation_roles

You can set an own role who should be able to add a type. Use an Tuple of Strings. Default and example for this value: `("Manager", "Owner", "Member")`.

documentation

You can add documention via this tag; it's better to use your UML tool's documentation field.

email

You can set the email project-wide with the `--email` commandline parameter (or in the config file). This tag allows you to overwrite it on a model level.

imports

A list of python import statements which will be placed at the top of the generated file. Use this to make new field and widget types available, for example. Note that in the generated code you will be able to enter additional import statements in a preserved code section near the top of the file. Prefer using the imports tagged value when it imports something that is directly used by another element in your model. You can have several import statements, one per line, or by adding several tagged values with the name `imports`.

label

Sets the readable name.

license

You can set the license project-wide with the `--license` commandline parameter (or in the config file). This tag allows you to overwrite it on a model level.

migrate_dynamic_view_fti

Migrates FTI of a type/class to CMFDynamicViewFTI. This works only if the class derives from an ATContentType, from ATCTMixIn or direct from CMFDynamicViewFTI. browserdefault.BrowserDefaultMixin.

module

Like `module_name`, it overwrites the name of the directory it'd be normally placed in.

module_name

Like `module`, it overwrites the name of the directory it'd be normally placed in.

read_permission

Defines archetypes fields read-permission. Use it together with workflow to control ability to view fields based on roles/permissions.

relation_implementation

Sets the type of implementation is used for an association: `basic` (used as default) for classic style archetypes references or `relations` for use of the `Relations` Product.

rename_after_creation

Setting this boolean value enables or disables explicit the after creation rename feature using `_at_rename_after_creation` class-attribute.

use_portal_factory

Setting this boolean value enables the registration of the type for use with portal_factory.

write_permission

Defines archetypes fields write-permission. Use it together with workflow to control ability to write data to a field based on roles/permissions.

package

association_class

You can use associations classes to store content on the association itself. The class used is specified by this setting. Don't forget to import the used class properly.

association_vocabulary

Switch, defaults to False. Needs Product `ATVocabularyManager`. Generates an empty vocabulary with the name of the relation.

author

You can set the author project-wide with the `--author` commandline parameter (or in the config file). This tag allows you to overwrite it on a package level.

copyright

You can set the copyright project-wide with the `--copyright` commandline parameter (or in the config file). This tag allows you to overwrite it on a package level.

creation_permission

Sets the creation permission for the class. Example: `Add portal content`.

creation_roles

You can set an own role who should be able to add a type. Use an Tuple of Strings. Default and example for this value: `("Manager", "Owner", "Member")`.

documentation

You can add documention via this tag; it's better to use your UML tool's documentation field.

email

You can set the email project-wide with the `--email` commandline parameter (or in the config file). This tag allows you to overwrite it on a package level.

imports

A list of python import statements which will be placed at the top of the generated file. Use this to make new field and widget types available, for example. Note that in the generated code you will be able to enter additional import statements in a preserved code section near the top of the file. Prefer using the imports tagged value when it imports something that is directly used by another element in your model. You can have several import statements, one per line, or by adding several tagged values with the name `imports`.

label

Sets the readable name.

license
: You can set the license project-wide with the `--license` commandline parameter (or in the config file). This tag allows you to overwrite it on a package level.

migrate_dynamic_view_fti
: Migrates FTI of a type/class to CMFDynamicViewFTI. This works only if the class derives from an ATContentType, from ATCTMixIn or direct from CMFDynamicViewFTI. browserdefault.BrowserDefaultMixin.

module
: Like `module_name`, it overwrites the name of the directory it'd be normally placed in.

module_name
: Like `module`, it overwrites the name of the directory it'd be normally placed in.

read_permission
: Defines archetypes fields read-permission. Use it together with workflow to control ability to view fields based on roles/permissions.

relation_implementation
: Sets the type of implementation is used for an association: `basic` (used as default) for classic style archetypes references or `relations` for use of the `Relations` Product.

rename_after_creation
: Setting this boolean value enables or disables explicit the after creation rename feature using `_at_rename_after_creation` class-attribute.

use_portal_factory
: Setting this boolean value enables the registration of the type for use with portal_factory.

write_permission
: Defines archetypes fields write-permission. Use it together with workflow to control ability to write data to a field based on roles/permissions.

portlet

author
: You can set the author project-wide with the `--author` commandline parameter (or in the config file). This tag allows you to overwrite it on a portlet level.

autoinstall
: Set to `left` or `right` to automatically install the portlet (a class with the stereotype `<<portlet>>`) with the product in the left or right slots, respectively. If it already exists in the slot it won't get overwritten.

copyright
: You can set the copyright project-wide with the `--copyright` commandline parameter (or in the config file). This tag allows you to overwrite it on a portlet level.

creation_permission
: Sets the creation permission for the class. Example: `Add portal content`.

creation_roles

You can set an own role who should be able to add a type. Use an Tuple of Strings. Default and example for this value: `("Manager", "Owner", "Member")`.

documentation

You can add documention via this tag; it's better to use your UML tool's documentation field.

email

You can set the email project-wide with the `--email` commandline parameter (or in the config file). This tag allows you to overwrite it on a portlet level.

imports

A list of python import statements which will be placed at the top of the generated file. Use this to make new field and widget types available, for example.

Note that in the generated code you will be able to enter additional import statements in a preserved code section near the top of the file.

Prefer using the imports tagged value when it imports something that is directly used by another element in your model.

You can have several import statements, one per line, or by adding several tagged values with the name `imports`.

label

Sets the readable name.

license

You can set the license project-wide with the `--license` commandline parameter (or in the config file). This tag allows you to overwrite it on a portlet level.

module

Like `module_name`, it overwrites the name of the directory it'd be normally placed in.

module_name

Like `module`, it overwrites the name of the directory it'd be normally placed in.

view

Set the name of the portlet. Defaults to the method name. This will be used as the name of the auto-created page template for the portlet.

state

documentation

You can add documention via this tag; it's better to use your UML tool's documentation field.

initial_state

Sets this state to be the initial state. This allows you to use a normal state in your UML diagram instead of the special round starting-state symbol.

label

Sets the readable name.

worklist

Attach objects in this state to the named worklist. An example of a worklist is the to-review list.

worklist:guard_permissions
Sets the permissions needed to be allowed to view the worklist. Default value is `Review portal content`.

state transition

documentation
You can add documention via this tag; it's better to use your UML tool's documentation field.

label
Sets the readable name.

trigger_type
Sets the trigger type, following what is defined by DCWorkflow: 0 : Automatic 1 : User Action (default) 2 : Workflow Method

tool

author
You can set the author project-wide with the `--author` commandline parameter (or in the config file). This tag allows you to overwrite it on a tool level.

autoinstall
Set to true (1) to automatically install the tool when your product is installed.

configlet
Set to true (1) to set up a configlet in the Plone control panel for your tool.

configlet:condition
A TALES expresson defining a condition which will be evaluated to determine whether the configlet should be displayed.

configlet:description
A description of the configlet.

configlet:icon
The name of an image file, which must be in your product's skin directory, used as the configlet icon.

configlet:permission
A permission which is required for the configlet to be displayed.

configlet:section
The section of the control panel where the configlet should be displayed. One of `Plone`, `Products` (default) or `Members`.

configlet:title
The name of the configlet.

configlet:view
The id of the view template to use when first opening the configlet. By default, the `view` action of the object is used (which is usually base_view)

copyright
You can set the copyright project-wide with the `--copyright` commandline parameter (or in the config file). This tag allows you to overwrite it on a tool level.

creation_permission
Sets the creation permission for the class.

Example: `Add portal content`.

creation_roles

You can set an own role who should be able to add a type. Use an Tuple of Strings. Default and example for this value: `(“Manager”, “Owner”, “Member”)`.

documentation

You can add documention via this tag; it’s better to use your UML tool’s documentation field.

email

You can set the email project-wide with the `--email` commandline parameter (or in the config file). This tag allows you to overwrite it on a tool level.

imports

A list of python import statements which will be placed at the top of the generated file. Use this to make new field and widget types available, for example. Note that in the generated code you will be able to enter additional import statements in a preserved code section near the top of the file. Prefer using the imports tagged value when it imports something that is directly used by another element in your model. You can have several import statements, one per line, or by adding several tagged values with the name `imports`.

label

Sets the readable name.

license

You can set the license project-wide with the `--license` commandline parameter (or in the config file). This tag allows you to overwrite it on a tool level.

module

Like `module_name`, it overwrites the name of the directory it’d be normally placed in.

module_name

Like `module`, it overwrites the name of the directory it’d be normally placed in.

tool_instance_name

The id to use for the tool. Defaults to `portal_<name>`, where <name> is the class name in lowercase.

toolicon

The name of an image file, which must be found in the skins directory of the product. This will be used to represent your tool in the Zope Management Interface.

Stereotype overview

Overview of all stereotypes you can use. (Note 2005-08-25: stereotypes are complete, but there are some empty descriptions). This pages is autogenerated with 'python UMLProfile.py'.

class

CMFMember
The class will be treated as a CMFMember member type. It will derive from CMFMember's Member class and be installed as a member data type. Identical to `<<member>>`.

archetype
Explicitly specify that a class represents an Archetypes type. This may be necessary if you are including a class as a base class for another class and ArchGenXML is unable to determine whether the parent class is an Archetype or not. Without knowing that the parent class in an Archetype, ArchGenXML cannot ensure that the parent's schema is available in the derived class.

btree
Like `<<folder>>`, it generates a folderish object. But it uses a BTree folder for support of large amounts of content. The same as `<<large>>`.

content_class
TODO

doc_testcase
Turns a class into a doctest class. It must subclass a `<<plone_testcase>>`.

field
TODO.

folder
Turns the class into a folderish object. When a UML class contains or aggregates other classes, it is automatically turned into a folder; this stereotype can be used to turn normal classes into folders, too.

hidden
Generate the class, but turn off `global_allow`, thereby making it unavailable in the portal by default. Note that if you use composition to specify that a type should be addable only inside another (folderish) type, then `global_allow` will be turned off automatically, and the type be made addable only inside the designated parent. (You can use aggregation instead of composition to make a type both globally addable and explicitly addable inside another folderish type).

interface_testcase
TODO

large
Like `<<folder>>`, it generates a folderish object. But it uses a BTree folder for support of large amounts of content. The same as `<<large>>`.

member

The class will be treated as a CMFMember member type. It will derive from CMFMember's Member class and be installed as a member data type. Identical to `<<CMFMember>>`.

mixin

Don't inherit automatically from `BaseContent` and so. This makes the class suitable as a mixin class.

odStub

Prevents a class/package/model from being generated. Same as `<<stub>>`.

ordered

For folderish types, include folder ordering support. This will allow the user to re-order items in the folder manually.

plone_testcase

Turns a class into the (needed) base class for all other `<<testcase>>` and `<<doc_testcase>>` classes inside a `<<test>>` package.

portal_tool

Turns the class into a portal tool.

python_class

Generate this class as a plain python class instead of as an archetypes class.

setup_testcase

Turns a class into a testcase for the setup, with pre-defined common checks.

stub

Prevents a class/package/model from being generated.

testcase

Turns a class into a testcase. It must subclass a `<<plone_testcase>>`. Adding an interface arrow to another class automatically adds that class's methods to the testfile for testing.

tool

Turns the class into a portal tool. Similar to `<<portal_tool>>`.

variable_schema

Include variable schema support in a content type by deriving from the VariableSchema mixin class.

vocabulary

TODO: Describe ATVocabularyManager support.

vocabulary_term

TODO: Describe ATVocabularyManager support.

widget

TODO.

dependency

value_class

Declares a class to be used as value class for a certain field class (see `<<field>>` stereotype)

method

action

Generate a CMF action which will be available on the object. The tagged values `action` (defaults to method name), `id` (defaults to method name), `category` (defaults to `object`), `label` (defaults to method name), `condition` (defaults to empty), and `permission` (defaults to empty) set on the method and mapped to the equivalent fields of any CMF action can be used to control the behaviour of the action.

form

Generate an action like with the `<<action>>` stereotype, but also copy an empty controller page template to the skins directory with the same name as the method and set this up as the target of the action. If the template already exists, it is not overwritten.

portlet

Create a simple portlet page template with the same name as the method. You can override the name by setting the `view` tagged value on the method. If you add a tagged value `autoinstall` and set it to `left` or `right`, the portlet will be automatically installed with your product in either the left or the right slot. If the page template already exists, it will not be overwritten.

This tutorial applies to:

Any version

Note on the PDF version of 28 November 2005

This document has been reformatted in Adobe InDesign CS2 based on an XMLish import of
http://members.plone.org/documentation/tutorial/archgenxml-getting-started/tutorial_full_view

Thanks to Joel Burton for quickly putting together the ‘tutorial_full_view’

There are still formatting ‘errors’ since the field and stereotype definitions have hard-coded line breaks. I have removed a bunch of them, but there are still many left.

Maybe I’ll get to those later, but in the interim, I hope this will be useful.

Peter Fraterdeus, semiotx
peterf@mac.com
http://www.fraterdeus.com