Atacama Large Millimeter / submillimeter Array

Alma Common Software Component Code Generation

Project Documentation

ALMA-BBB-CCC-A-DDD

Version: B

Status: Draft

2010-31-01

Keywords: Alma Common Software Component Java Code Generator

	Name	Date	Signature
Prepared	Alexis Tejeda	2010-04-23	
Approved			
Released			

Change Record

Issue/Rev.	Date	Section/Parag. affected	Reason/Initiation/Documents/Remarks
A	2010-02-10	All	Creation, A.TEJEDA.
B	2010-04-23	All	Update all document, ATEJEDA.

Table of Contents

List of Figures

1 INTRODUCTION

1.1 Purpose

The creation of components based over Alma Common Software (ACS), can be a very repetitive task, for example the configuration of the CDB, definition of the IDL files and the classes creation, always is pretty much the same development process. Here is when the Model Driven Development (MDD) becomes a powerfull tool for the code generation, in this case, based on defined templates for UML class diagrams, part of this work is already done in the thesis [1] of Nicolas Troncoso.

In order to improve development time, code refactoring and the software engineering (at pattern desings level) for the component development over ACS, will continue the work already done by Nicolas Troncoso proposed in his thesis [1], please check thesis in Referenced Documents section in this document.

The purpose of this document is to explain the new features, the new development,how the generator works and how the generation 'generates' the code from UML model/metamodel/meta-metamodels, as well as providing guidelines for the future development and extensions of the component code generator.

1.2 Scope

The content of the document is for developers, software engineers and managers, to allow them to use in a easy and good way the component generator for ACS, to create in a fast way code ready for the implementation.

It is assumed that the reader has a good command in the creation of components for ACS and understand the functioning of ACS and knows Java OOP paradigm.

1.3 Reference Documents

[1] ACS Component Code Generation Framework
https://csrg.inf.utfsm.cl/twiki4/pub/ACS/AlmaTheses/thesis-ntroncos09.pdf

[2] Eclipse Modeling Framework Documents
http://www.eclipse.org/modeling/emf/docs/

[3] Agile Model Driven Development with UML 2
Cambridge University Press, 2004 ISBN#: 0-521-54018-6

[4] Open Architecture Ware Reference
http://www.openarchitectureware.org/pub/documentation/4.3.1/

1.4 Abbreviations and Acronyms

- ACS : Alma Common Software.
- EMF : Eclipse Modeling Framework, a framework of Eclipse to create plug-ins, Eclipse Applications for code generation base in Model Driven.
- NC : Notification Channels of ACS.
- oAW : Open Architecture Ware, a free open architecture for code generation.
- XMI : XML Metadata Interchange, standard for exchanging metadata information via Extensible Markup Language (XML).
- UML : Unified Modeling Language, is a standardized general-purpose modeling language in the field of software engineering.
- MDD: Model Driven Development
- OOP : Object Oriented Programming

1.5 Stylistic Conventions

The following styles are used:

bold
In the text, to highlight words.

italic
In the text, for parts that have to be substituted with the real content before typing. Also used to highlight words or section names.

`teletype`
In the text, for commands, filenames, pre/suffixes as they have to be typed. Also used for file content examples.

`<name>`
In examples, for parts that have to be substituted with the real content before typing.

`<<stereotype>>`
Stereotypes used in the code generator.

2 PROJECT OVERVIEW

2.1 Previous Work

The work done before this project is related to the thesis[1] of Nicolas Troncoso, which gives the very beginning to the ACS Code Generation based on UML metamodels, this project will cover and continue the work already done , proposed in the thesis as a 'Future Work', like the implementation of inheritance, more complex models, separation of implemented and generated classes and have a full implemented code generator for ACS.

2.2 Project Objectives

The main goal/objective in the project is have a full Standalone Java Component Code Generator for Alma Common Software under Eclipse Modeling Framework umbrella, this, is divided in specific objectives:

- Project based in Open Architecture Ware 5:
  All the project has to be based on Open Architecture Ware 5, now under the EMF umbrella.
- Managment of complex UML Models:
  In complex UML models, the use of stereotypes that allow to filter the models to generate, this mean that a class will have a stereotype to identify if this class will be generated or not, this is for not re-write classes already generated. This objective is considered a critical factor for the success of the project.
- Alma Common Software - Notification Channels Support:
  A full code generation support for ACS Notification Channels, generate all necessary code from a UML model that represents the association between the Supplier and Consumers, the generator must be able to generate the ready-to-implement code for Notification Channels. This objective is considered a critical factor for the success of the project.
- Standalone Code Generator:
  The development of the generator should be under the EMF, however one of the critical requirements is have the generator as Standalone, this means that the generator have to work outside the Eclipse IDE dinamicly with any UML model/metamodel specified by command line. The generator have to generate all the classes, plus empty skelletons for the user implementation. or regenerate only the basic classes, this mean only generates the templates for user modification. This objective is considered a critical factor for the success of the project.
- The generated code must be Java:
  At first, all the code generated must be in Java, later, if the time allow, then implement the code generator templates for C++ and Python, it's more importante in this project have a full support Java for the code generated than a semi-implementation of all programming languages.
- Separation between the generated classes and implemented classes:
  Complete the separation between the generated classes and the implemented classes,

this mean the code generator has to know how to separate the implemented classes and the already generated classesfor the code generation for not overwriting.

- Inheritance implementation:
  Actually, the implementation in the generator of model-inheritance is not fully supported, the developers should define how implement this model behavior in the code generator without affect all work already done.

2.3 Project Scope

The project scope cover the development of the Code Generator for ACS, initially the project will only generate the Java code implementation from a UML model under the XMI2.0 standard, in which, there are three importants points defined as critical and project milestones.

- Standalone Java Component Code Generator.
- Notification Channels and Inheritance Support for UML models.
- Code Generator based on Open Architecture Ware 5.

Each one of this points are explained more consistently in Goals and Objectives sections in this document.

2.4 Roles and Responsibilities

- Alexis Tejeda (UCN) Deployment Manager, Requirements Reviewer, Architecture Reviewer, Configuration Manager, Change Control Manager .
- Gianluca Chiozzi (ESO) Client, End User, Requirements Reviewer, Change Control Manager.
- Jorge Ibsen (ESO / JAO Computing Manager) Project Manager, Requirements Reviewer.
- Nicolas Troncoso (JAO/AUI) Project Manager, Requirements Reviewer, Architecture Reviewer.

3 DESIGN OVERVIEW

3.1 EMF - Open Architecture Ware 5

The component code generator for Alma Common Software initially created before, was supported on Open Architecture Ware 4, since is a requirement have the code generator based on Open Architecture Ware 5, the previous work was migrated to oaW 5, this means that the code generator is based now on Eclipse Modeling Framework (EMF), mainly in Xtend and Xpand subprojects.

EMF supports many types of models like Ecore (EMF native models), UML2 models, or, in this case meta-metamodels in XMI2.0 format. Actually the generator support a XMI2.0 meta-metamodels, mostly of this models are created and exported from MagicDraw UML Diagram Tool, MagicDraw generate the necessary XMI files for later generate the code components to be implemented by the final user. Is important to specify that the models needs a profile file, which defines the stereotypes for use in the generator, where they differentiate the many characteristics or custom features in the UML model.

3.1.1 Stereotypes

In UML models, the use of stereotypes helps to distinguish the classes that must be generated and how must be generated. This stereotypes are defined in the UML profile file, which is created when the UML model is exported from MagicDraw.
One of each classes may have a stereotype defined to know how the code must be generated or specify a certain feature desired, i.e.:

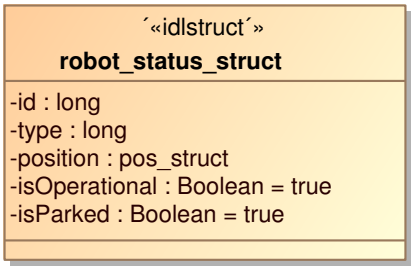

Figure 1: Example class with a stereotype to be generated as a IDL struct

The class `robot_status_struct` has the `<<IDLStruct>>` stereotype, then the generator call the IDLStruct template and generate the code of an IDL file of ACS and implement such struct in the IDL file..

There exist three main categories of stereotypes:

- Class stereotype : applies to classes. stereotype.
- Property stereotype : applies to class level variables.
- Operation stereotype : applies to a class method.

The stereotypes helps to differentiate how should the code must be generated. Currently the generator has several types of (only class level) stereotypes defined below:

- `<<enumeration>>`: UML class is implemented as an enumeration and is generated in a common IDL file.
- `<<idlstruct>>`: UML class is implemented as an IDL struct generated in a common IDL file.
- `<<characteristic>>`: The UML class is implemented as a Characteristic Component of ACS.
- `<<override>>`: If the UML class is inherited, the generator overrides the parent methods (of the UML model) in the actual class, this only is applied to Java classes and not to the IDL interfaces and is implemented as a ACS component.
- `<<alloverride>>`: Same as the override stereotype, if the class has a multi-inherited-level, the generator overrides all methods inherited in all levels in the UML model and is implemented as a ACS component, this only is applied to Java classes and not to the IDL interfaces.
- `<<container>>`: The UML class is defined as a ACS container, the generator reads a property called 'language' to know the languageof the container (i.e.: language = java) and generate the necessary code to implement this container.
- `<<ncconfig>>`: If the UML class has this stereotype, the generator reads the channels to implement (properties in the UML class) in the code, this channels are defined in the common IDL file of the code generated.
- `<<ncsupplier>>`: The UML class is implemented as a ACS component event supplier and use a common channel that has the same name of the project.
- `<<ncconsumer>>`: The UML class is implemented as a ACS component event consumer and use a common channel that has the same name of the project.
- `<<nchybrid>>`: The UML class is implemented as a ACS component event supplier-consumer and use a common channel that has the same name of the project.
- `<<void>>`: If this stereotype is implemented, the generator will void generate the class.

The stereotypes above, are defined in the profile file of the XMI exported UML model and can be mixed in the UML model, i.e: a class can have the `<<ncconsumer>>` and `<<override>>` to generate an ACS component that override his parent methods and is implemented as a consumer, but, not all stereotypes can be mixed, the developer must have a certain common logic to mix the stereotypes, this logic is related to the knowing of ACS component development.

Below, a table of how can mix the stereotypes:

	idlstruct	characteristic	override	alloverride	container	ncconfig	ncsupplier	ncconsumer	nchybrid
idlstruct	-								
characteristic		-	yes	yes			yes	yes	yes
override		yes	-				yes	yes	yes
alloverride		yes		-			yes	yes	yes
container					-				
ncconfig						-			
ncsupplier		yes	yes	yes			-		
ncconsumer		yes	yes	yes				-	
nchybrid		yes	yes	yes					-

3.1.2 Xtend/Xpand

Since owA was migrated to EMF, the code generator is based on a mixture of EMF subprojects(Xpand/Xtend/Xtext) , the EMF subprojects which are used:

- Xtend : provides the possibility to define rich libraries of independent operations and non-invasive metamodel extensions based on either Java methods or oAW expressions. Those libraries can be referenced from all other textual languages, that are based on the expressions framework, in this case Xpand.
- Xpand : A programming language which allow to define the templates of the generator and controll the output generation.

3.1.3 Workflow File Configuration

The workflow, is a EMF XML configuration file that controll the workflow of the generator, in which, where configured the paths of UML exported models, output folder, templates used, Java code beautifiers and the package name definition for the generated code.

The path of the model to generate, the generated code output folder path and the UML profile file path are specified using dynamic workflow propertys by ${myVariable} sintax. These properties can be configured dynamically in the Java program (using String Hashmap) that call the generator, or by command line.

Also the templates to use in the generation are specified in the configuration file as 'Workflow Components'.

The components must have defined :

- Output path : the output path for the generated file, this is specified by a global property in the workflow configuration as ${ouputFolderURI} property.
- UML profile : this is specified by a global property in the workflow configuration as ${profileFileURI} property, the generator only support maximum three profile files.

- Template : the template to use.
- Beautifiers : XML or Java code beautifiers.
- VetoStrategy : only is specified if the component will use a Veto Strategy, see 'Generator Optimization' section for more info about this.
- File encoding : by default UTF-8.

An example configuration for a component, in this case a java class files component generator :

```
<component id="genjava" class="org.eclipse.xpand2.Generator"
 skipOnErrors="true">
<fileEncoding value="UTF-8" />
    <metaModel idRef="default_profile"/>
    <expand value="templates::java::JavaRoot::Root FOR model"/>
    <outlet path="${ouputFolderURI}">
        <vetoStrategy
         class="cl.alma.acs.ccg.vetostrategy.ACSCCGVetoStrategy" />
        <postprocessor
         class="org.eclipse.xpand2.output.JavaBeautifier"/>
    </outlet>
    <prSrcPaths value="${ouputFolderURI}"/>
</component>
```

In the project, exists three workflow files, a Java, C++ and Python. Python and C++ workflow files are ready to be implemented, only Java workflow is full implemented for the generator.

The workflow file is based in oaW 5, this means that the file presents many changes from his previous version.

3.1.4 Template Files

Template files controll the output code generation. Each class in the UML model is analyzed by the template file, the template check the stereotype of the class and generate the output file. The templates are based in the Xpand programming language [4].
All templates are encoded in UTF-8 by the use of 'guillimets' [4].

3.1.5 Xtend Util Helper

In the generator templates folder exists a Xtend file, this file is a helper for the templates files, in which, are defined helper functions like if a class is inherited from other class. More info about the functions, see 'Quick Reference User Manual' in this document.

3.2 Generator Optimization

With a simple model (20 classes) the generator can take about 8 or 10 seconds to generate the code in a Dual CPU@1.73GHz with 2048 MB RAM, but, if the model is more complex, then

the generation can take longer, this is no problem at all, the problem comes when the code must be generated again for any reason, like add a method in a class or add another class in the model. This is a problem, because, every time we want re-generate the code, will take the same time as the first generation even if there are no changes in the model (same code to generate.)

To fix this, the use of EMF Veto Strategy is implemented in the components workflow file configuration. This strategy is class that implements a EMF Veto interface class, in which each file to be generated, is analyzed if presents any changes, if there any changes, the files is generated again, if not, the file is not generated again.

This strategy improves the generation time in regeneration, complex models, and model refactoring.

3.3 Source Code Structure

The generator is packaged in a JAR Java file, under the reverse domain name cl.alma.acs.ccg which follow the Java package folder structure, the source code use Ant or Eclipse Ant to compile the project.

```
cl.alma.acs.ccg
|-- bin
|   |-- ACSCCG.class
|   |-- cl
|   |   ‘-- alma
|   |       ‘-- acs
|   |           ‘-- ccg
|   |               |-- mwe
|   |               |   |-- CppWorkflow.mwe
|   |               |   |-- JavaWorkflow.mwe
|   |               |   ‘-- PythonWorkflow.mwe
|   |               |-- strategy
|   |               |   |-- CodeCppGeneration.class
|   |               |   |-- CodeJavaGeneration.class
|   |               |   |-- CodePythonGeneration.class
|   |               |   |-- ContextCodeGeneration.class
|   |               |   ‘-- ICodeGenerationStrategy.class
|   |               |-- vetostrategy
|   |               |   ‘-- ACSCCGVetoStrategy.class
|   |               ‘-- vo
|   |                   ‘-- VOGenerator.class
|   ‘-- templates
|       |-- cpp
|       |-- java
|       |   |-- CDB.xpt
|       |   |-- IDLCommonRoot.xpt
|       |   |-- IDLCommon.xpt
|       |   |-- IDLComponent.xpt
```

```
|       |   |-- JavaCharacteristicComponent.xpt
|       |   |-- JavaComponent.xpt
|       |   |-- JavaContainer.xpt
|       |   |-- JavaHelper.xpt
|       |   |-- JavaInheritanceAllOverride.xpt
|       |   |-- JavaInheritanceOverride.xpt
|       |   |-- JavaInterfaceImplements.xpt
|       |   |-- JavaInterface.xpt
|       |   |-- JavaRoot.xpt
|       |   |-- Makefile.xpt
|       |   |-- SchemaRoot.xpt
|       |   |-- Schema.xpt
|       |   `-- xJavaUtil.ext
|       `-- python
|-- build.properties
|-- build.xml
|-- lib
|   `-- commons-cli-1.2.jar
|-- META-INF
|   `-- MANIFEST.MF
|-- project
`-- src
    |-- ACSCCG.java
    |-- cl
    |   `-- alma
    |       `-- acs
    |           `-- ccg
    |               |-- mwe
    |               |   |-- CppWorkflow.mwe
    |               |   |-- JavaWorkflow.mwe
    |               |   `-- PythonWorkflow.mwe
    |               |-- strategy
    |               |   |-- CodeCppGeneration.java
    |               |   |-- CodeJavaGeneration.java
    |               |   |-- CodePythonGeneration.java
    |               |   |-- ContextCodeGeneration.java
    |               |   `-- ICodeGenerationStrategy.java
    |               |-- vetostrategy
    |               |   `-- ACSCCGVetoStrategy.java
    |               `-- vo
    |                   `-- VOGenerator.java
    `-- templates
        |-- cpp
        |-- java
        |   |-- CDB.xpt
        |   |-- IDLCommonRoot.xpt
        |   |-- IDLCommon.xpt
        |   |-- IDLComponent.xpt
        |   |-- JavaCharacteristicComponent.xpt
```

```
|   |-- JavaComponent.xpt
|   |-- JavaContainer.xpt
|   |-- JavaHelper.xpt
|   |-- JavaInheritanceAllOverride.xpt
|   |-- JavaInheritanceOverride.xpt
|   |-- JavaInterfaceImplements.xpt
|   |-- JavaInterface.xpt
|   |-- JavaRoot.xpt
|   |-- Makefile.xpt
|   |-- SchemaRoot.xpt
|   |-- Schema.xpt
|   `-- xJavaUtil.ext
`-- python
```

3.3.1 Class Diagram

In the class diagram, the only classes not developed (in the project) are those which belongs to the packages `org.eclipse.xpand2.output` and `org.eclipse.emf.mew.core`.

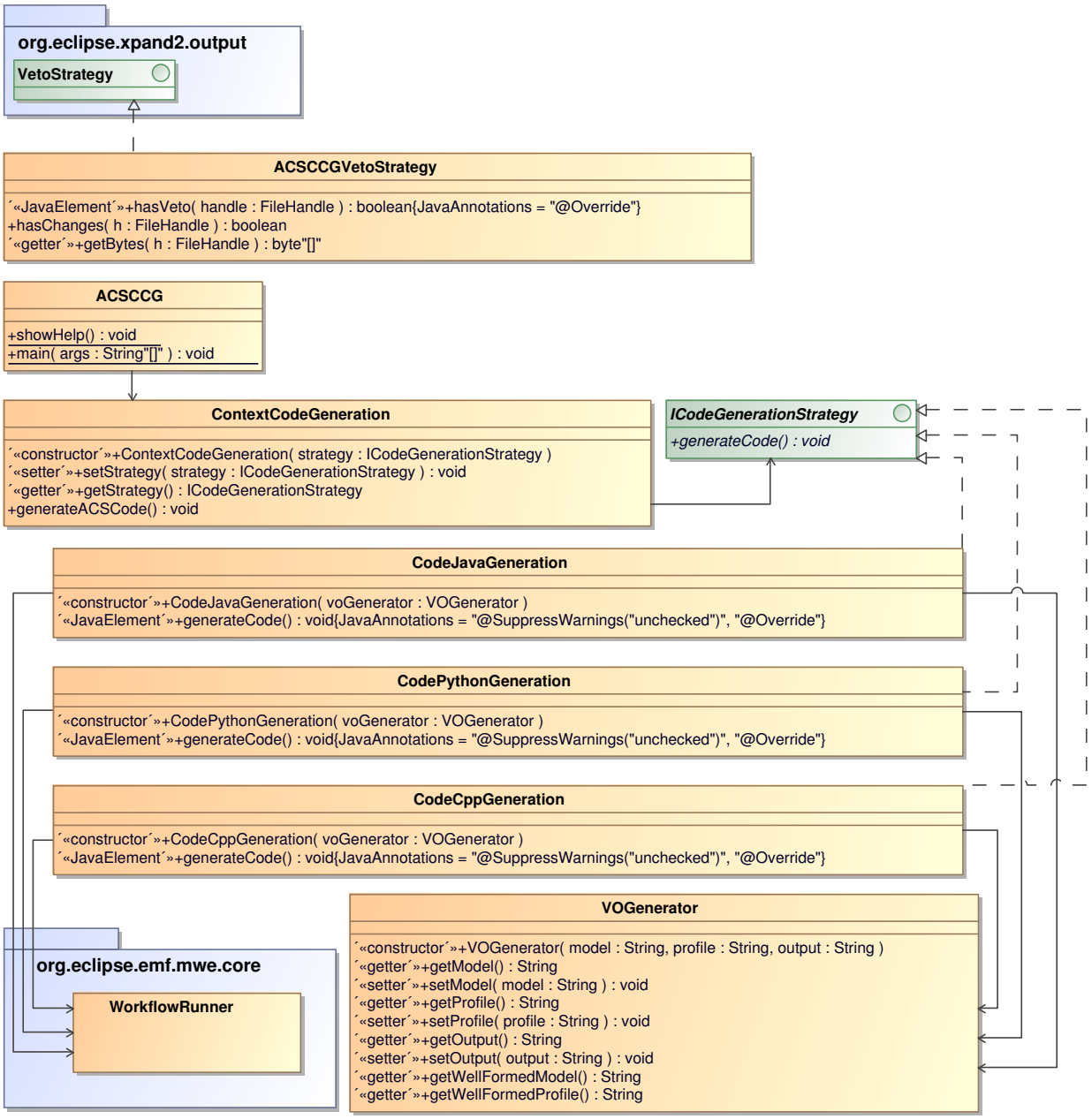

Figure 2: Component Code Generator Class Diagram

3.3.1.1 Strategy Pattern

A Strategy Pattern (Policy Pattern) was implemented in the component code generator to generate the code for each programming languaje in ACS whithout change top levels algorithms, due to the scope of the project, only Java strategy is full implemented.

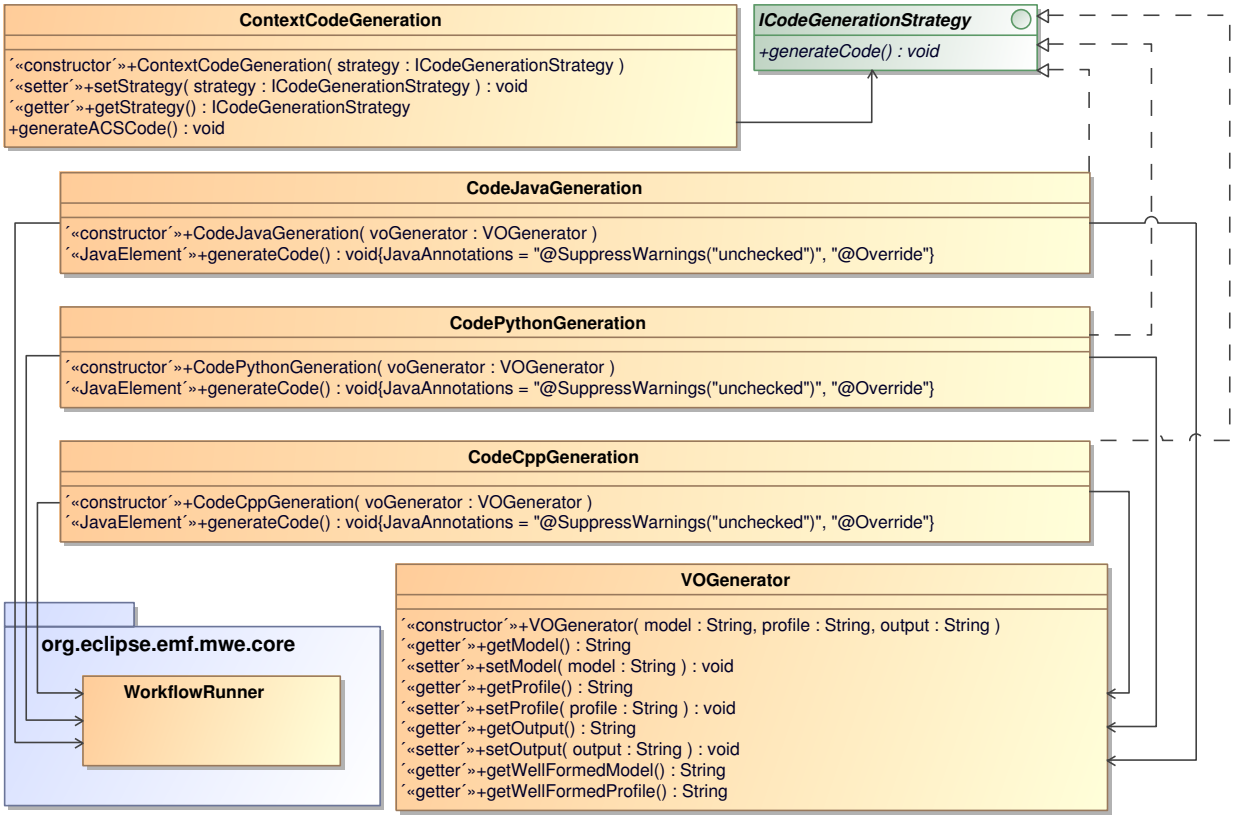

Figure 3: Strategy Pattern

The `ContextCodeGeneration` is the main class to be called for implement the generator in plugins, Java programs or other enviroments.

3.3.1.2 EMF Veto Strategy

The class `ACSCCGVetoStrategy` is called by WorkflowRunner in runtime of the component code generator, for more info about Veto, see Generator Optimization section in this document.

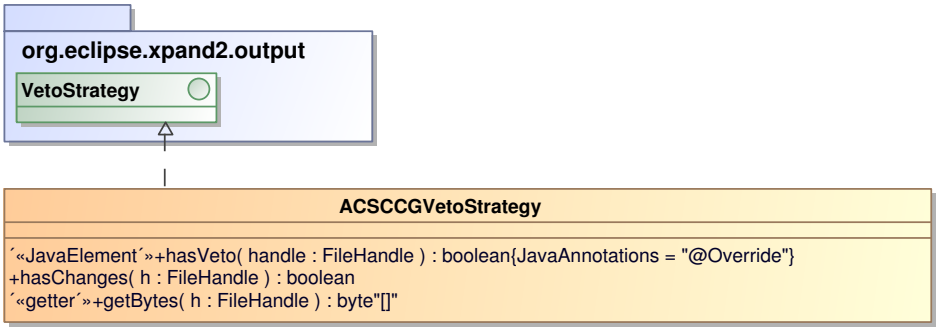

Figure 4: EMF Veto Strategy

4 COMPONENT CODE GENERATOR FEATURES

In this section will be describe the main features of the code generator, such as the inheritance, notification channels, a standalone generator, code regeneration strategies.

4.1 A Stand-alone Generator

The component code generator was designed to be a standalone application, this means, can be executed in any O.S. as a Java Program, JAR Package or Eclipse Plugin outside of EMF or Eclipse EMF project.

The generator supports three ways to be a standalone :

- JAR Package : The binary files are packaged in a JAR Java file using Ant or Eclipse Ant.
- Eclipse Plugin : A simple Eclipse plugin as a Jar file.
- Command Line : A Java command line program.

Also, the generator contains all the packages to be executed without EMF and can be imported to other environments running Java.

All the binarys can be dowloaded, for this, see the 'Project Paths' in this document.

4.2 Basic Component Modeling

The basic component model contains a at least a Container and a Component class also a IDL struct or enumeration class.

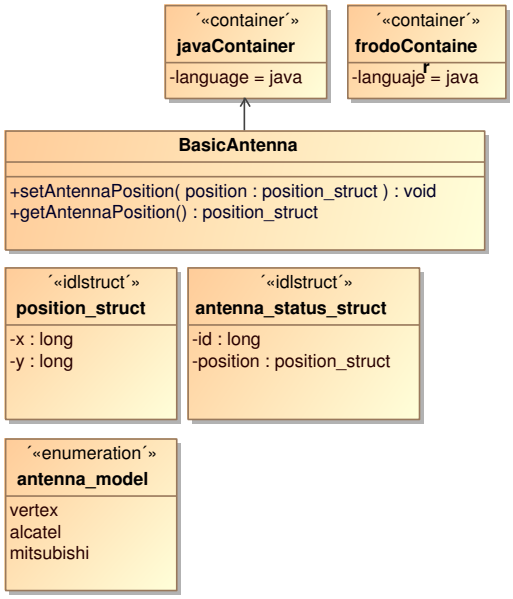

Figure 5: Basic Model Definition

This example can be downloaded from: http://acsccg.googlecode.com/files/example1.tar.gz

4.2.1 Container Definition

Every model should have defined at least one container. This is defined by the stereotype `<<container>>` and the class must have defined a property called 'language', the value of this property is the programming language that the container will support (cpp, java, python). All the containers defined are generated in the CDB configuration in the generated code.

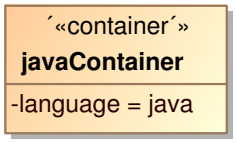

Figure 6: Container Definition

In the example above, the class `javaContainer` has the stereotype `<<container>>` and the languaje for this container is `java`. Below, is the code generated for the containers, in this case the `javaContainer`.

```
test/
'-- CDB
    '-- MACI
        |-- Components
        |   '-- Components.xml
        |-- Containers
        |   '-- javaContainer
        |       '-- javaContainer.xml
        '-- Managers
            '-- Manager
                '-- Manager.xml
```

And the `javaContainer.xml` code:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Container
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="urn:schemas-cosylab-com:Container:1.0"
    xmlns:log="urn:schemas-cosylab-com:LoggingConfig:1.0"
    ImplLang="java"
    >
    <Autoload>
     </Autoload>
     <LoggingConfig>
        <log:_ Name="jacorb@javaContainer"
        minLogLevel="5"
        minLogLevelLocal="5"/>
    </LoggingConfig>
```

```
</Container>
```

4.2.2 Component Definition

Every class without a stereotype, the generator will recognize the class as a component. Every class must be related to a container class, if the class is not related to a container, the generator will not config the component in the CDB. All component classes that are related to a container the generator will generate:

- The source component Java code (with his methods).
- The source java helper class.
- The IDL interface (with component methods).
- The CDB configuration for the component.
- The Makefile is generated with all IDL interfaces which represents each class component in the model.

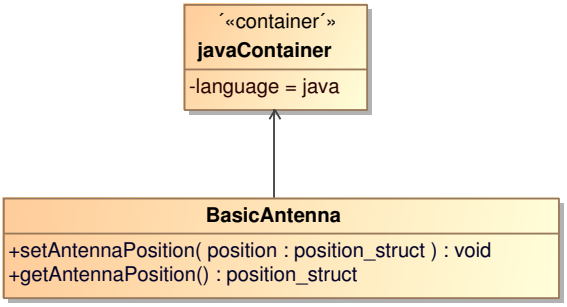

Figure 7: Basic Component Definition

In the example above the class `BasicAntenna` is a ACS component and is related to the `javaContainer` the name of the Java classes are prefixed by 'Base' also the IDL interfaces generated are prefixed by 'Base' and example of this the `BasicAntenna` class is generated as `BasicAntennaBase.java` class.

4.2.3 IDL Structs Definition

The IDL structs are defined in a class with the stereotype `<<idlstruct>>`. In the example below the class `<<antenna_status_struct>>` has two property that the generator will implement.

The code generated for that struct is defined in a common IDL interface, this IDL file has the name of the project, in this case `example1Common.idl`, an example of the code generated:

```
#ifndef example3_IDL
#define example3_IDL
#pragma prefix "alma"
```

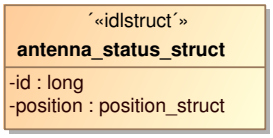

Figure 8: IDL Struct Definition

```
module example3
{
    struct position_struct {
        long x;
        long y;
    };
    typedef sequence<position_struct> position_struct_seq;

    struct antenna_status_struct {
        long id;
        position_struct position;
    };
    typedef sequence<antenna_status_struct> antenna_status_struct_seq;
};
#endif //example3_IDL
```

The code above also implement a `position_struct` which is used by `antenna_status_struct`.

4.2.4 Enumerations Definition

The enumerations are defined as a class with the `<<enumeration>>` stereotype in the model. The code for the enumerations is generated in the IDL common file, t, in this case `example1Common.idl`.

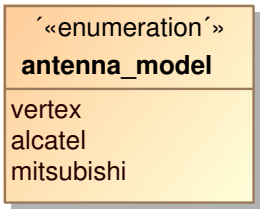

Figure 9: Basic Enumeration Definition

An example of the code generated:

```
....
module example3
{
    enum antenna_model { vertex, alcatel, mitsubishi };
```

....

4.2.5 Code Generated

For convention, the model name has to be in a reverse domain name syntax, (see the examples) i.e.:

```
cl.alma.acsccg.example1
```

The generator will generate the source code using the reverse domain name syntax for generate the Java code. An example of the code generated for the Figure 5 with the model name as `cl.alma.acsccg.example1` (this can be viewed in the folder structure).

```
example3/
|-- idl
|   |-- BasicAntennaBase.idl
|   `-- example1Common.idl
|-- src
|   |-- cl
|   |   `-- alma
|   |       `-- acsccg
|   |           `-- example1
|   |               |-- BasicAntennaBaseHelper.java
|   |               `-- BasicAntennaBase.java
|   `-- Makefile
`-- test
    `-- CDB
        `-- MACI
            |-- Components
            |   `-- Components.xml
            |-- Containers
             |-- javaContainer
            |   |   `-- javaContainer.xml
            |   `-- frodoContainer
            |       `-- frodoContainer.xml
            `-- Managers
                `-- Manager
                    `-- Manager.xml
```

Also is generated the project Makefile, in which is defined the name of the Jar file and the IDL interfaces to compile. The name of the Jar file is the last name in the model name.

4.3 Java Interfaces

The generator was designed to support Java interfaces in the UML model, this interfaces are implemented by the generator writing the methods in the component generated code.

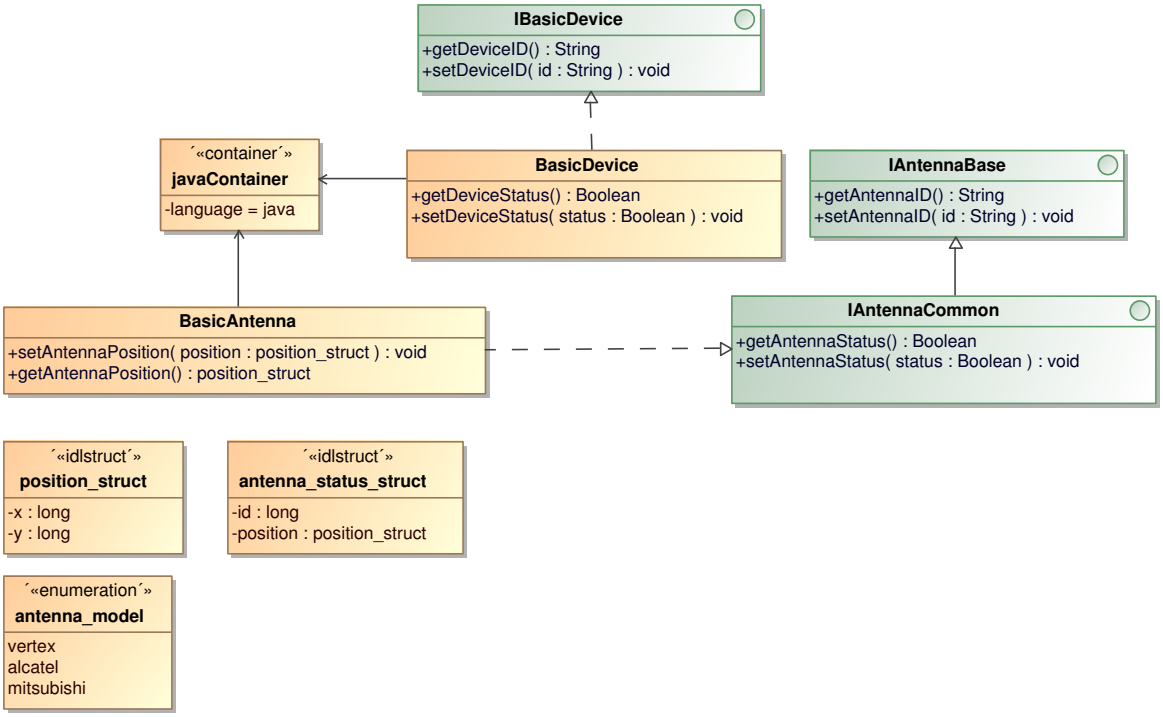

Figure 10: Interfaces.

In the figure above, the class `BasicAntenna` implements the Interface `IAntennaCommon` and `IAntennaCommon` is extended from `IAntennaBase`, the generator also can implement the inheritance in Interfaces. The code generated for the `BasicAntenna` component implements the all the methods of all implemented interfaces, if the interface is extended in one or multiple levels, the generator implements all methods of the interface inheritance (Java OOP constraints) and the IDL file implement this methods to.

Code generated, the component implementes the java interface defined in the model above.

```
...
public class BasicAntennaBase
      implements
         ComponentLifecycle,
              BasicAntennaBaseOperations,
              IAntennaCommon {
...
```

And all methods from `IAntennaCommon`, `IAntennaBase` are implemented in the component.

```
...
```

```
    /*
     * Implements the Interface methods.
     */
    @Override
    public boolean getAntennaStatus() {...

    @Override
    public void setAntennaStatus(boolean status) {...

    @Override
    public String getAntennaID() {...

    @Override
    public void setAntennaID(String id) {....
 ...
```

Also the IDL file of the component is implemented with this methods to:

```
...
#ifndef BasicAntenna_IDL
#define BasicAntenna_IDL
#include <acscomponent.idl>
#include <example3Common.idl>

#pragma prefix "alma"

module example3
{
    interface BasicAntennaBase : ACS::ACSComponent
    {
        void setAntennaPosition(in position_struct position);
        position_struct getAntennaPosition();
        boolean getAntennaStatus();
        void setAntennaStatus(in boolean status);
        string getAntennaID();
        void setAntennaID(in string id);
    };
};
#endif /* example3_IDL */
...
```

This example can be downloaded from: http://acsccg.googlecode.com/files/example3.tar.gz

4.4 Inheritance

4.4.1 Multiple Level

The generator is designed to support inheritance in one ore more inherited levels, with the ability to override the inherited methods from the parenet, or, override all methods inherited

in all inherited levels.
This can be viewed in figure 11.

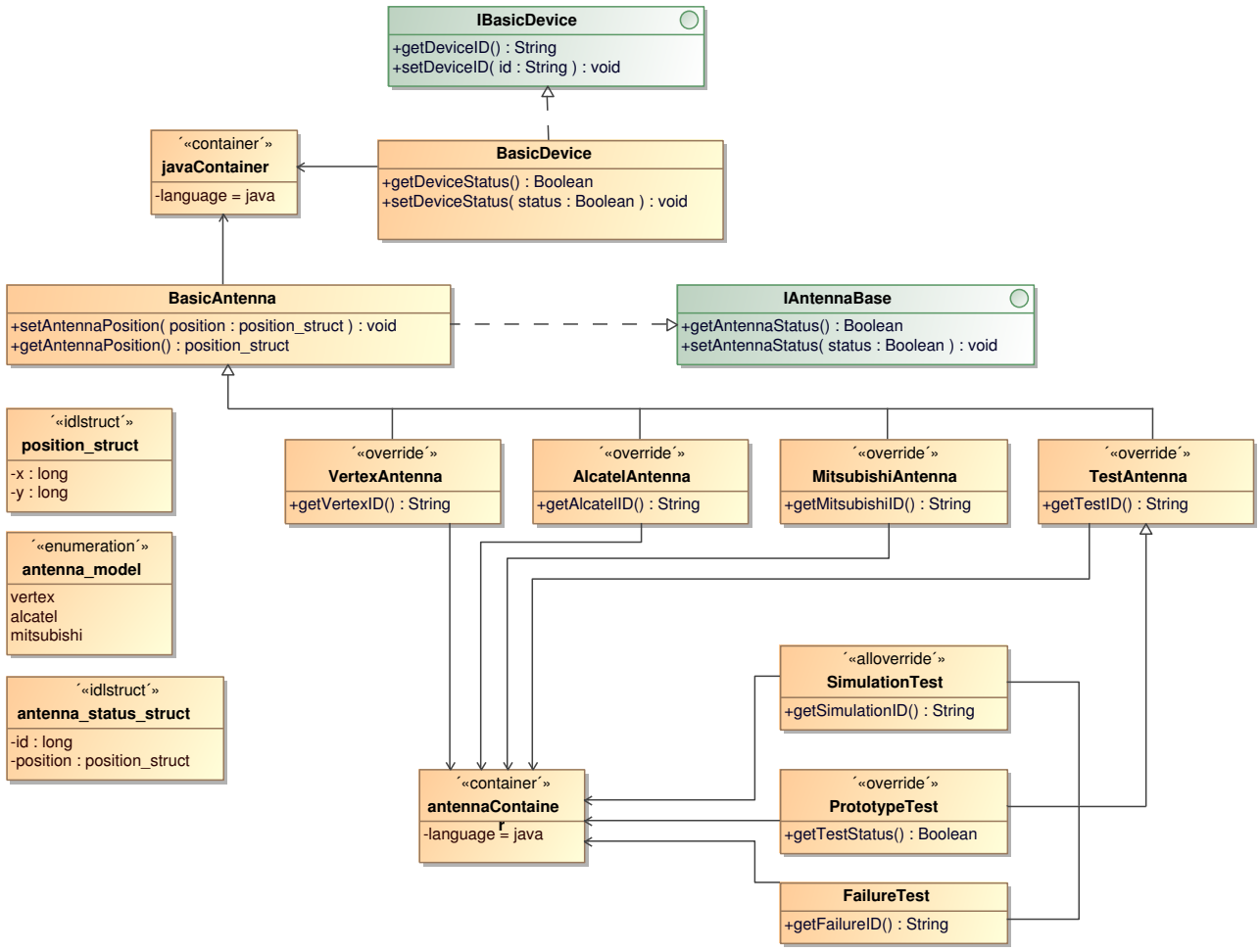

Figure 11: Inheritance in the generator

This example can be downloaded from: http://acsccg.googlecode.com/files/example4.tar.gz

In the example above the classes with the `<<override>>` stereotype will override the methods from his parents in the Java generated code, and the classes with the `<<alloverride>>` will override all methods inherited in all leves, i.e.: the class `SimulationTest` will have his own methods, the `TestAntenna` methods, the `BasicAntenna` methods (with the methods of the implemented interface `IAntennaBase`).
If the component class has a parent class, the `initialize` method of the component will call the `initialize` method of the parent class, an example of this is:

```
...
public class VertexAntennaBase extends BasicAntennaBase...

public void initialize(ContainerServices containerServices) {

m_containerServices = containerServices;
```

```
super.initialize(containerServices);
...
```

The IDL files also implement the inheritance, an example of this:

```
...
module example4
{
     interface AlcatelAntennaBase : BasicAntennaBase
    {
        string getAlcatelID();
        void setAntennaPosition(in position_struct position);
        position_struct getAntennaPosition();
...
```

Also if the class component is not inherited from other component or other class, the IDL interface is extended from `ACS::ACSComponent`, the AlcatelAntennaBase is extended from BasicAntennaBase.

4.5 Characteristic Component

A class with the stereotype `<<Characteristic>>`, if is extended from other class, the generator will not implement the inheritance in the Java code, because the Java OOP paradigm not support multiple inheritance (parallel inheritance) and the class with `<<Characteristic>>`, the generator will generate the class already extended from ACS class 'CharacteristicComponentImpl', also is generated the schema configuration .

4.6 Notification Channel

The ACS Notification Channel also is supported in the UML model.

This example can be downloaded from: http://acsccg.googlecode.com/files/example6.tar.gz

In the example above exists a class with the `<<ncconfig>>` stereotype, this class define the channels to use in the project, in the example they are defined three channels `statuschannel`, `positionchannel` and `infochannel` generated in the common IDL interface, in this case `example6Common.idl`, should only be one class with this stereotype.
Also when this class is defined in the model, the generator define a common or default channel called in this case `defaultexample6channel`, the `example6` in the channel name, is from the reverse domain name syntax on model naming and a IDL struct for test the channel implementation, this struct is called `testMessageBlockEvent`, an example of the code generated :

```
...
//Channels
const string CHANNELNAME_DEFAULTEXAMPLE6CHANNEL = "defaultexample6channel";
const string CHANNELNAME_STATUSCHANNEL = "statuschannel";
const string CHANNELNAME_POSITIONCHANNEL = "positionchannel";
const string CHANNELNAME_INFOCHANNEL = "infochannel";
```

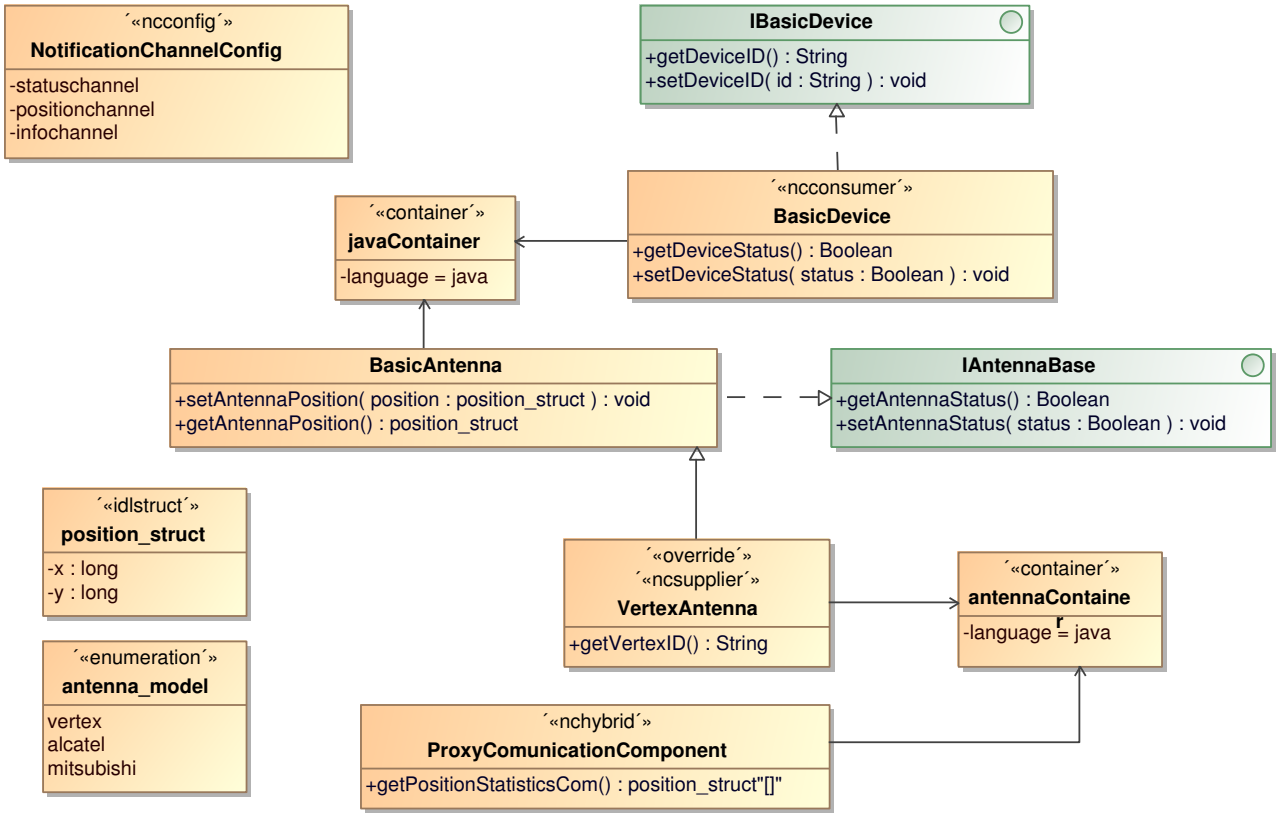

Figure 12: Inheritance in the generator

```
 //Test Struct Event for the notification channels
struct testMessageBlockEvent
{
     double randomID;
     string message;
 };
  typedef sequence<testMessageBlockEvent> testMessageBlockEvent_seq;
...
```

Also in the example, they are classes with the `<<ncsupplier>>`, `<<ncconsumer>>` and the `<<nchybrid>>`:

4.6.1 ncsupplier

The classes with `<<ncsupplier>>` stereotype, will supply events in the default channel defined by the generator, in this case the channel is called:
`const string CHANNELNAME_DEFAULTEXAMPLE6CHANNEL="defaultexample6channel";`
Also the generator will implement the necessary methods to supply events (a default event called `testMessageBlockEvent`) in the channel, an example of the code generated:

```
...

import alma.acs.nc.SimpleSupplier;
import alma.example6.testMessageBlockEvent;

...

    public void initialize(ContainerServices containerServices) {

        m_containerServices = containerServices;

        super.initialize(containerServices);

        m_logger = m_containerServices.getLogger();
        m_logger.info("initialize() called...");

        try {
            // Instantiate our supplier
                m_supplier = new SimpleSupplier(
                    alma.example6.CHANNELNAME_DEFAULTEXAMPLE6CHANNEL.value,
                m_containerServices);
        } catch (Exception e) {
            //throw new ComponentLifecycleException("failed to create
        SimpleSupplier for channel " +
        alma.example6.CHANNELNAME_DEFAULTEXAMPLE6CHANNEL.value, e);
        }

....

    public void sendEvents() throws CouldntPerformActionEx {
       m_logger.info("Now sending  testMessageBlockEvent events...");
        try {
            testMessageBlockEvent event = new testMessageBlockEvent(Math
                .random(), "Event");
            m_supplier.publishEvent(event);
        } catch (Throwable thr) {
            m_logger
                .log(Level.WARNING,
                    "failed to send testMessageBlockEvent. Will not try
                    again."); throw (new AcsJCouldntPerformActionEx(thr))
                    .toCouldntPerformActionEx();
        }
    }
```

4.6.2 ncconsumer

The classes with <<ncconsumer>> stereotype, will consume events in the default channel defined by the generator, in this case the channel is called:

```
const string CHANNELNAME_DEFAULTEXAMPLE6CHANNEL="defaultexample6channel";
```

Also the generator will implement the necessary methods to consume events (a default event called `testMessageBlockEvent`) in the channel, an example of the code generated:

```
...
    private Consumer m_consumer;
    ....
        try {
            m_consumer = new Consumer(
                alma.example6.CHANNELNAME_DEFAULTEXAMPLE6CHANNEL.value,
                    m_containerServices);
                //Subscribe to a domain and event type.
            m_consumer.addSubscription(
                alma.example6.testMessageBlockEvent.class, this);
                m_consumer.consumerReady();
                m_logger
                    .info(" BasicDevice  is waiting for
                    'testMessageBlockEvent'events.");
                    } catch (Exception e) {
                if (m_consumer != null) {
            m_consumer.disconnect();
           }
            /*
             if uncomment the next line, add  throws
             ComponentLifecycleException, but first check if is
             inhitered class, can not be override the method.
             the generator put the right code. */
             throw new ComponentLifecycleException(
             "failed to connect as an event consumer to channel " +
             alma.example6.CHANNELNAME_DEFAULTEXAMPLE6CHANNEL.value);
             }
    ....

...
```

4.6.3 nchybrid

The classes with `<<nchybrid>>` will consume and supply events in the default channel defined by the generator, this class will implement the supply and consume methods, variables and librarys to test the component. This methods and variables are the same of the examples above.

For the three stereotypes, if the components has not a parent class, the generator will implement an ACS throw exception in the `initialize` method of the component.

4.7 Implemented Code vs Generated Code

In every model driven development and code generators, is important how make a strategy to separate the code already generated with the code implemented over the generated code. They are three ways solve this.

4.7.1 EMF Veto Strategy and Inheritance

The Veto EMF strategy (see Generator Optimization section) is a good solution using inheritance to implement new things, but, must be create a new class for every change in our code to void override the whole code already generated, so, this is not the best solution, but, it works.

4.7.2 GAP Desing Pattern

The generation GAP desing pattern provides separate the code with all hand-modifications implemented in sub-classes, this mean that the core classes are generated only once and the hand-made implementations, are extended as subclases from core classes.

4.7.3 Protected Areas

Xpand, provides protected areas to our generated code, this mean, that certain areas with the protected tag with a unique autegenerated ID, can be modified by the developer without loosing the hand-modifications over the generated code when the code is re-generated (even if classes are changed in the model). i.e.: This code went generated with protected areas.

```
1. public void getSimulationList() {
2. /*PROTECTED REGION ID(getSimulationList) ENABLED START*/
3.    //Implementation Method here!
4. /*PROTECTED REGION END*/
5. }
```

```
1. Method definition
2. Start Protected Area
3. Method hand implementation
4. End Protected Area
```

Protected areas are specified in generator templates, the generator analize the generated code, check the code with the template, and protect the area from the regeneration. If the protected region is not in the templates, the generator will void the area.
In methods, only the implementation is protected and for the code that scape from UML model every class has a protected area for custom imports, variables and methods.