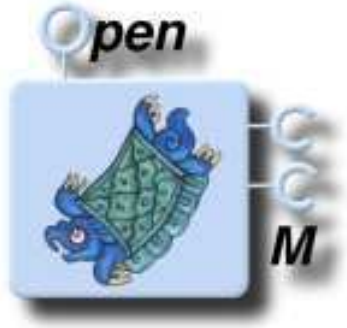

The ObjectWeb Consortium

Developer guide

Inside OpenCCM

AUTHORS:

Areski Flissi (CNRS-LIFL)

CONTRIBUTORS:

OpenCCM Team

Released: Mars 2003

Status: Final Draft

Version: 1.0

TABLE OF CONTENTS

1 INTRODUCTION 6

1.1 GOALS 6
1.2 TARGET AUDIENCE 6
1.3 DOCUMENT CONVENTION 6

2 INSIDE OPENCCM PLATFORM AND PRODUCTION CHAIN 7

2.1 OVERVIEW OF THE OPENCCM PRODUCTION CHAIN 7
2.2 THE PARSER IMPLEMENTATION SOURCE FILES 10
2.3 THE ABSTRACT SYNTAX TREE (AST) 13
2.4 IMPLEMENTATION OF THE OPENCCM GENERATORS 15
2.4.1 The Basic Generator 16
2.4.2 The OpenCCM OMG IDL Generator 17
2.4.3 The OpenCCM PSDL Generator 18
2.4.4 The OpenCCM CIDL Generator 18
2.4.5 The OpenCCM CIF Generator 19
2.5 OPENCCM PRODUCTION CHAIN COMMANDS 20
2.5.1 About OpenCCM Command Scripts 22
2.5.2 Starting the OpenCCM's OMG IDL3 Repository (IR3) 25
2.5.3 Feeding the OpenCCM IR3 with an OMG IDL3 file 25
2.5.4 Generator for equivalent OMG IDL 2.4 mappings from an OMG IDL3 file .. 26
2.5.5 OpenCCM IDL3 Generator Command 27
2.5.6 OpenCCM PSDL Generator Command 27
2.5.7 OpenCCM CIDL Generator Command 27
2.5.8 CIF and Java Component Segmented Implementation Skeletons Generator .. 28
2.5.9 Java Container Classes Generator 28
2.5.10 Java Component Implementation Templates Generator 29
2.5.11 XMI 1.1 Generator 29
2.6 IDL, XML, DTD AND COMMAND SCRIPTS SOURCE DIRECTORIES 31
2.6.1 OpenCCM IDL Sources 31
2.6.2 OpenCCM Unix and Windows Command Scripts Sources 32
2.6.3 OpenCCM DTD Sources 33
2.6.4 OpenCCM XML Configuration File Sources 34

3 OPENCCM RUNTIME 35

3.1 THE OPENCCM COMPONENTS RUNTIME 35
3.2 THE OPENCCM CONTAINERS RUNTIME 37
3.3 THE OPENCCM CIF RUNTIME 39
3.4 A CCM APPLICATION RUNNING, HOW DOES IT WORK ? 41
3.4.1 The Generated Java Extended Skeletons and Interceptors 42
3.4.2 The Generated IDL to Java Mappings 44
3.4.3 Implementation Classes and Dependencies with OpenCCM Runtime 58
3.5 THE INSIDE OPENCCM BIG PICTURE 61

4 OPENCCM DEPLOYMENT AND EXECUTION INFRASTRUCTURE 63

4.1 INSTALLATION OF THE OPENCCM DEPLOYMENT INFRASTRUCTURE 63
4.2 STARTING THE NAME SERVICE OF THE USED ORB 63
4.3 THE JAVA COMPONENT SERVER 63
4.4 WHAT SHOULD BE DONE IN A DEPLOYMENT PROCESS: A SIMPLE EXAMPLE 65

4.4.1 Initializing the ORB and the OpenCCM Runtime 65
4.4.2 Obtaining the Name Service 65
4.4.3 Obtaining Component Servers 65
4.4.4 Obtaining Container Homes and Archives 66
4.4.5 Installing Archives 66
4.4.6 Installing a Container on each Server 67
4.4.7 Installing Homes 67
4.4.8 Creating and Configuring Components 68
4.4.9 Connecting Components 69
4.4.10 Configuration Completion 69
4.5 THE OPENCCM DEPLOY TOOL 70
4.5.1 Inside the Deployment Tool Process 70
4.5.2 Writing XML Meta Information 71
4.6 DEPLOYING AND EXECUTING THE SIMPLE CCM CLIENT / SERVER EXAMPLE 72

5 COMPILATION AND INSTALLATION OF OPENCCM PLATFORM 74

5.1 COMPILATION OF OPENCCM PLATFORM 74
5.2 INSTALLATION OF OPENCCM PLATFORM 74

6 ANNEXES 76

6.1 OPENCCM GENERATORS CLASS DIAGRAM 76
6.1.1 The Basic Generator Class Diagram 76
6.1.2 The IDL Generator Class Diagram 77
6.1.3 The PSDL Generator Class Diagram 77
6.1.4 The CIDL Generator Class Diagram 78
6.1.5 The CIF Generator Class Diagram 78
6.2 TRACE SERVICE IN OPENCCM (USING MONOLOG FRAMEWORK, OBJECTWEB) 79
6.2.1 Monolog Framework concepts 79
6.2.2 Use of trace in OpenCCM code base 80
6.2.3 Monolog Configuration 82
6.2.4 Local TraceService Class diagram 83

TABLE OF FIGURES

FIGURE 1 – OVERVIEW OF THE OPENCCM PRODUCTION CHAIN 7
FIGURE 2 – THE OPENCCM DISTRIBUTION ARCHITECTURE 8
FIGURE 3 - OPENCCM SOURCE DIRECTORY CONTENT 9
FIGURE 4 - THE PARSER ARCHITECTURE 10
FIGURE 5 - JAVA IMPLEMENTATION OF THE PARSER 11
FIGURE 6 - THE ABSTRACT SYNTAX TREE 13
FIGURE 7 - JAVA IMPLEMENTATION OF THE AST 14
FIGURE 8 - THE OPENCCM GENERATORS ARCHITECTURE 15
FIGURE 9 - THE OPENCCM GENERATORS IMPLEMENTATION 16
FIGURE 10 - THE BASIC GENERATOR 17
FIGURE 11 - THE OMG IDL3 AND IDL2 GENERATORS 17
FIGURE 12 - THE OPENCCM CIDL GENERATOR IMPLEMENTATION 19
FIGURE 13 - THE OPENCCM CIF GENERATOR IMPLEMENTATION 19
FIGURE 14 - OPENCCM PRODUCTION CHAIN COMMANDS 20
FIGURE 15 - JAVA IMPLEMENTATION OF OPENCCM COMMANDS 21
FIGURE 16 - INTERFACES FOR OPENCCM COMMAND 21
FIGURE 17 - OPENCCM COMMAND CLASSES DIAGRAM 22
FIGURE 18 - THE LAUNCHER DTD 24
FIGURE 19 - XML CONFIGURATION FILES FOR ORB AND PRODUCTION CHAIN 25
FIGURE 20 – THE XML CONFIGURATION FILE OF OPENCCM'S IR3INSTALL 25
FIGURE 21 - THE XML CONFIGURATION FILE THE OPENCCM'S IR3FEED 26
FIGURE 22 - THE XML CONFIGURATION FILE THE OPENCCM'S IR3TOIDL2 GENERATOR 27
FIGURE 23 – OMG IDL, PSDL AND CIDL DEPENDENCIES 28
FIGURE 24 - THE XMI GENERATOR PACKAGE 30
FIGURE 25 - THE XMI 1.1 UML GENERATOR COMMAND 31
FIGURE 26 - OPENCCM IDL SOURCES 31
FIGURE 27 - OPENCCM UNIX AND WINDOWS COMMAND SCRIPTS SOURCES 33
FIGURE 28 – OPENCCM DTD FILES 33
FIGURE 29 - XML CONFIGURATION FILE SOURCES 34
FIGURE 30 - THE ORG.OBJECTWEB.CCM.COMPONENTS PACKAGE 35
FIGURE 31 - THE CCMOBJECTIMPL CLASS DIAGRAM OF OPENCCM COMPONENTS RUNTIME 36
FIGURE 32 - THE CONTAINER ARCHITECTURE 37
FIGURE 33 - THE ORG.OBJECTWEB.CCM.RUNTIME.CONTAINERS PACKAGE 38
FIGURE 34 - THE OPENCCM CIF RUNTIME PACKAGE 39
FIGURE 35 - CLASS DIAGRAM OF THE CIF RUNTIME 40
FIGURE 36 - CIF RUNTIME AND COMPONENT EXECUTOR SKELETONS DEPENDENCIES 41
FIGURE 37 - OUR SIMPLE CCM CLIENT / SERVER EXAMPLE 41
FIGURE 38 - OPENCCM GENERATED JAVA SKELETONS FOR CLIENT AND SERVER COMPONENTS . 42
FIGURE 39 - THE GENERATED JAVA EXTENDED SKELETON CLASSES DIAGRAM 44
FIGURE 40 - OMG IDL2.X MAPPINGS FROM CLIENT VIEW POINT 45
FIGURE 41 - CLIENT-SIDE IDL MAPPINGS FOR THE "CLIENT" COMPONENT 46
FIGURE 42 - CLIENT-SIDE IDL MAPPINGS FOR THE "SERVER" COMPONENT 47
FIGURE 43 - COMPONENT CLASS DIAGRAM FROM CLIENT VIEW POINT 48
FIGURE 44 - HOME CLASS DIAGRAM FROM CLIENT VIEW POINT 49
FIGURE 45 - OMG IDL2.X MAPPINGS FROM SERVER VIEW POINT 50
FIGURE 46 - SERVER-SIDE IDL MAPPINGS FOR THE "CLIENT" COMPONENT 52
FIGURE 47 - SERVER-SIDE IDL MAPPINGS FOR THE "SERVER" COMPONENT 53
FIGURE 48 - CLIENT-SIDE AND SERVER-SIDE CIDL MAPPINGS 55

FIGURE 49 - COMPONENT CLASS DIAGRAM FROM SERVER VIEW POINT 56
FIGURE 50 - HOME CLASS DIAGRAM FROM SERVER VIEW POINT 56
FIGURE 51 - GENERATED JAVA MAPPINGS FOR OUR SIMPLE CCM APPLICATION 58
FIGURE 52 - CLASS DIAGRAM FOR GENERATED AND COMPONENT IMPLEMENTATION CLASSES 60
FIGURE 53 - INSIDE OPENCCM BIG PICTURE 61
FIGURE 54 - THE GENERATED OPENCCM CONFIGURATION DIRECTORIES 63
FIGURE 55 - OPENCCM GENERATED COMPONENT SERVER CONFIGURATION FILES 64
FIGURE 56 - DEPLOYMENT IMPLEMENTATION PACKAGE 64
FIGURE 57 - SEQUENCE DIAGRAM OF THE AUTOMATED DEPLOYMENT PROCESS 70
FIGURE 58 - REFERENCING COMPONENT SOFWARE DESCRIPTORS IN CAD 72
FIGURE 59 - DEFINING PARTITIONING FOR DISTRIBUTED DEPLOYMENT 73
FIGURE 60 – DEFINING THE CONNECTIONS 73
FIGURE 61 - THE BUILD DIRECTORY OF INSTALLED OPENCCM PLATFORM 75
FIGURE 62 - BASIC GENERATOR CLASS DIAGRAM 76
FIGURE 63 - IDL GENERATOR CLASS DIAGRAM 77
FIGURE 64 - PSDL GENERATOR CLASS DIAGRAM 77
FIGURE 65 - CIDL GENERATOR CLASS DIAGRAM 78
FIGURE 66 - CIF GENERATOR CLASS DIAGRAM 78
FIGURE 67 - ORG.OBJECTWEB.UTIL.MONOLOG.PROVIDER.API 84
FIGURE 68 - ORG.OBJECTWEB.UTIL.MONOLOG.PROVIDER.LIB.TOPICS 85
FIGURE 69 – THE ORG.OBJECTWEB.CORBA.TRACE.PI PACKAGE 86
FIGURE 70 – OVERVIEW OF THE TRACE SERVICE 87

1 INTRODUCTION

This document is a developer guide describing the OpenCCM platform. OpenCCM is the first public available open source implementation of the Object Management Group's CORBA Component Model (OMG CCM, see the specification at http://www.omg.org/cgi-bin/doc?formal/2002-06-65). OpenCCM allows users to design, implement, compile, deploy and execute CCM compliant applications.

1.1 Goals

Through the OpenCCM platform architecture and source files, we are going to describe in this guide:

- The OpenCCM compilation, development and production chain:
  - The OMG IDL3, PSDL and CIDL compilers
  - The Abstract Syntax Tree (AST) and the OMG IDL3 Repository (IR3)
  - The IDL, CIDL, CIF and PSDL generators
  - CCM's OMG IDL and CIDL mappings
  - Java container generator (skeletons, interceptors, etc)
  - Java implementation template generator
  - XMI 1.1 UML documents generator
- The OpenCCM deployment infrastructure: support of the `ComponentInstallation`, `ComponentServer`, and `Container` interfaces of the CORBA Components specification, the deployment tool for Component Assembly Descriptor (CAD) XML descriptors, etc.
- The OpenCCM Runtime: the Java runtime library for Java Components, the Java Component server to deploy and execute Java components (*org.objectweb.ccm.Components*, *org.objectweb.ccm.Containers* and *org.objectweb.ccm.runtime.cif* packages).
- Others OpenCCM packages and their goal
- Results of the compilation and installation of OpenCCM

1.2 Target audience

The target audience for this tutorial includes all OpenCCM developers.

1.3 Document Convention

Description: Times New Roman:12

Example or source code: `Courier New:10`

2 INSIDE OPENCCM PLATFORM AND PRODUCTION CHAIN

2.1 Overview of the OpenCCM production chain

Input of the OpenCCM production chain be various kind of descriptions. This includes:

- CORBA components and interfaces definitions *via* OMG IDL3 (*Interface Definition Language*) files,
- Persistent states *via* OMG PSDL (*Persistent State Definition Language*) files,
- CORBA component's implementation structure *via* OMG CIDL (*Component Implementation Definition Language*) files.

The following figure shows a global view of the OpenCCM production and generation chain:

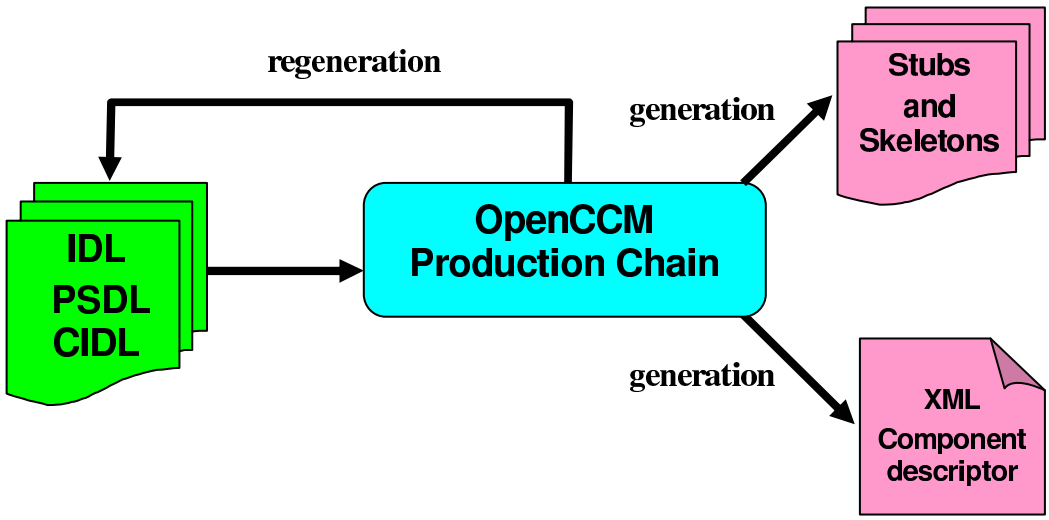

Figure 1 – Overview of the OpenCCM Production Chain

IDL / PSDL / CIDL compiler allows the generation of two kind of files:

- descriptions files, *i.e.*
  - IDL3, PSDL and CIDL regeneration
  - IDL2 equivalent mappings of IDL3 descriptions, and
  - IDL2 equivalent interfaces for CIDL descriptions (based on the CIF specification),
- runtime required files, *i.e.*
  - Java OpenCCM skeletons (component skeletons, container classes, CIF interfaces and component executor implementation skeletons), and
  - Java implementation patterns (on which users can start to write its Java implementation files, *i.e.* the business code).

In order to realize compilation and generation, several software elements are provided by the OpenCCM production chain:

- The Parser
- The Abstract Syntax Tree (AST)
- The Interface Repository (IR)
- Generators

In the next part of the document, we are going to detail the OpenCCM platform architecture for each part of production chain. OpenCCM platform is composed of many directories. Each part of the distribution contributes to the OpenCCM production chain.
The OpenCCM platform distribution architecture is the following:

```
OpenCCM
+---config
¦   +---BES-5.0.2
¦   +---JacORB-1.4
¦   +---OpenORB-1.2.1
¦   +---OpenORB-1.3.0
¦   +---ORBacus-4.1
+---demo
¦   +---common
¦   +---demo1
¦   +---demo2
¦   +---demo3
¦   +---dinner
¦   +---hello
+---doc
+---externals
¦   +---ant
¦   +---cpp
¦   +---ots
¦   +---velocity
¦   +---winprocess
¦   +---xml_dtd
+---src
¦   +---doxygen
¦   +---dtd
¦   +---idl
¦   +---java
¦   +---unix
¦   +---windows
¦   +---winprocess
¦   +---xml
+---test
```

Figure 2 – The OpenCCM Distribution Architecture

Before compilation and installation of the platform, the OpenCCM distribution contains the following directories: *config*, *demo*, *doc*, *externals*, *src* and *test*. Other directories are generated by the compilation process (for instance *ORBacus-4.1*, obviously depending of the installed user's ORB) and installation process (*build*, containing all necessary scripts and Java archive files).

Let us have a look at OpenCCM source files (the *src* directory):

```
+---src
¦   +---doxygen
¦   +---dtd
¦   +---idl
¦   +---java
¦   ¦   +---org
¦   ¦       +---objectweb
¦   ¦           +---ccm
¦   ¦           +---corba
¦   ¦           ¦   +---ast
¦   ¦           ¦   +---command
¦   ¦           ¦   +---generator
¦   ¦           ¦   +---parser
¦   ¦           ¦   +---util
¦   ¦           +---util
¦   +---unix
¦   +---windows
¦   +---winprocess
¦   +---xml
```

Figure 3 - OpenCCM Source Directory Content

The *java* directory contains the *org.objectweb.corba* OpenCCM package which provides implementation for the Parser, the AST and generators.

2.2 The Parser Implementation Source files

The following figure details the global architecture of the OpenCCM Parser:

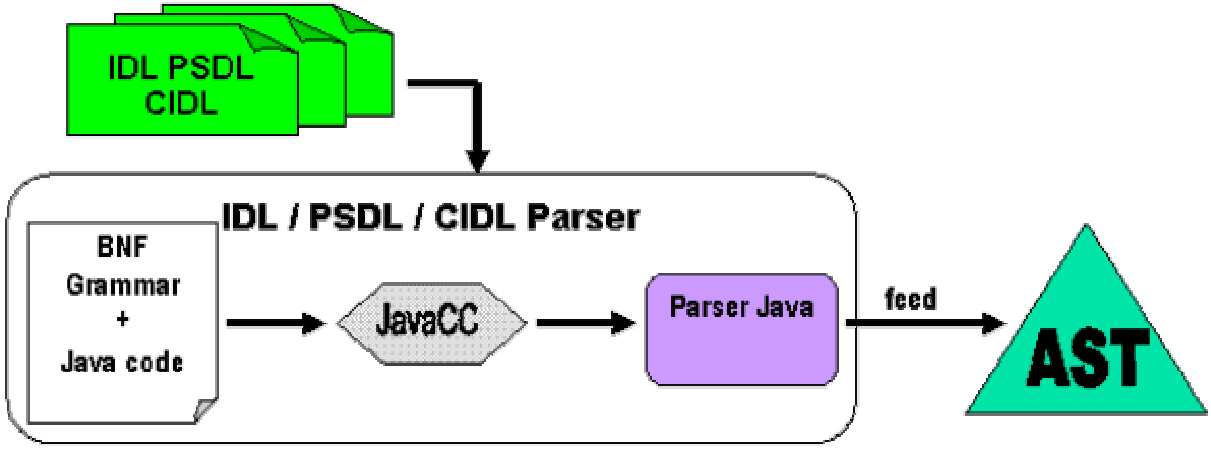

Figure 4 - The Parser Architecture

The parser uses the JavaCC tool (http://www.webgain.com/products/java_cc/) to generate the Java parser files, using a Backus Naur Form (BNF) grammar and Java code. It checks syntax and creates declarations in the Abstract Syntax Tree (AST). The AST is a mirror of parsed declarations of IDL, PSDL and CIDL files.

The following figure shows the files used by the Parser (generated Java files and main *Parser.jj* file) in the *org.objectweb.corba.parser* package.

```
+---src
|   +---doxygen
|   +---dtd
|   +---idl
|   +---java
|   |   \---org
|   |       \---objectweb
|   |           +---ccm
|   |           +---corba
|   |           |   +---ast
|   |           |   +---command
|   |           |   +---generator
|   |           |   +---parser
|   |           |   |   \---lib
|   |           |   |           DeclaratorList.java
|   |           |   |           ErrorManager.java
|   |           |   |           ParseException.java
|   |           |   |           Parser.java
|   |           |   |           Parser.jj
|   |           |   |           ParserConstants.java
|   |           |   |           ParserTokenManager.java
|   |           |   |           SimpleCharStream.java
|   |           |   |           Token.java
|   |           |   |           TokenMgrError.java
|   |           |   |
|   |           |   \---util
|   |           \---util
|   +---unix
|   +---windows
|   +---winprocess
|   \---xml
```

Figure 5 - Java Implementation of the Parser

The *lib* directory contains a main *Parser.jj* file which allows, using JavaCC tool, to generate Java parser files. As explained in the first part of this document, the parser allows to feed the AST by using IDL3, PSDL and CIDL description files.

Here is an example of a BNF grammar:

```
/**
 * (115) <component_header> ::= "component" <identifier>
 *                                    [ <component_inheritance_spec> ]
 *                                    [ <supported_interface_spec> ]
 */
ComponentDecl component_header() :
{
    String name;
    ComponentDecl component = null;
}
{
  "component" name=identifier()
    {
        try
        {
            component = currentScope_.declareComponent(name);
        }
        catch (org.omg.CORBA.SystemException exc)
        {
            error(exc);
        }
    }
  [ component_inheritance_spec(component) ]
  [ supported_interface_spec(component.getSupportedInterfaceList()) ]
    {
        return component;
    }
}
```

The first thing to do is to give a name for the rule (*component_header()* here). Our production rule is similar to a Java method and so can take some parameters and return a specific type (in this case, this rule takes no parameter and return a *ComponentDecl* type). Next, we declare variables used by the rule (here *name* and *component*). Then we have to write the body of the rule:

- *"component"* keyword means that the parser expect this specific word in the file.
- Inside the body of the rule, we insert Java code to add processing. In this particular rule, using *component = currentScope_.declareComponent(name)* allows to declare the component in the AST.

Mainly, the body of a rule is composed of expected specific keywords, calls of other declared rules and Java code for processing.

2.3 The Abstract Syntax Tree (AST)

The Abstract Syntax Tree (called AST in the next part of the document) is part of the OpenCCM production chain allowing to make the link between the Java Parser described below and the Interface Repository (IR).

The following figure details links between the declarations put in the AST and the IR:

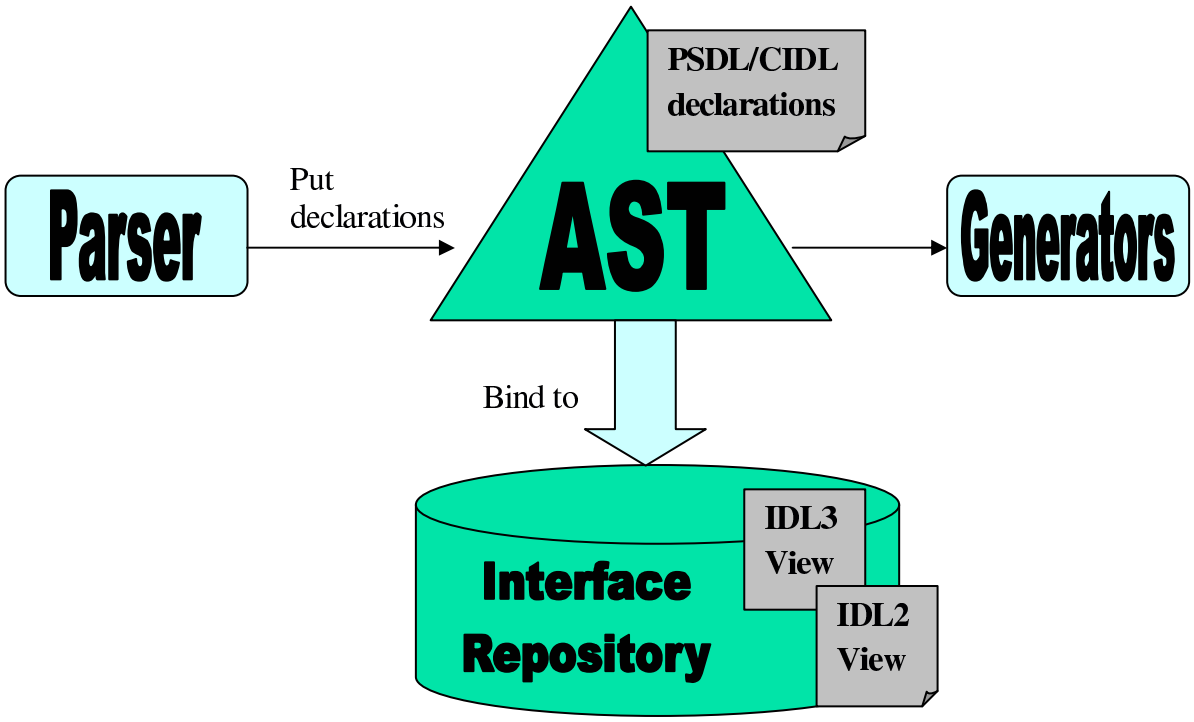

Figure 6 - The Abstract Syntax Tree

The AST is bound to the Interface Repository. An IDL3 declaration put in the AST is automatically registered in the interface reposirory (IR). The AST can be fed from the IR for IDL3 declarations.

It can also contain PSDL and CIDL declarations and providing IDL interface to access these declarations.

Java implementation files for the AST are the following:

```
+---src
|   +---doxygen
|   +---dtd
|   +---idl
|   +---java
|   |   \---org
|   |       \---objectweb
|   |           +---ccm
|   |           +---corba
|   |           |   +---ast
|   |           |   |   +---api
|   |           |   |   |       ASTError.java
|   |           |   |   \---lib
|   |           |   |           AbstractInterfaceDeclImpl.java
|   |           |   |           AbstractStorageHomeDeclImpl.java
|   |           |   |           AbstractStorageHomeListImpl.java
|   |           |   |           AbstractStorageTypeDeclImpl.java
|   |           |   |           AbstractStorageTypeListImpl.java
|   |           |   |           AliasDeclImpl.java
|   |           |   |           AnyValueImpl.java
|   |           |   |           AttributeDeclImpl.java
|   |           |   |           CidlScopeImpl.java
|   |           |   |           ComponentBaseImpl.java
|   |           |   |           ComponentDeclImpl.java
|   |           |   |           CompositionDeclImpl.java
|   |           |   |           ConstantDeclImpl.java
|   |           |   |           ConsumesDeclImpl.java
|   |           |   |           DeclarationImpl.java
|   |           |   |           DeclarationKindImpl.java
|   |           |   |           DeclarationListImpl.java
|   |           |   |           ...
|   |           |   |
|   |           |   +---command
|   |           |   +---generator
|   |           |   +---parser
|   |           |   \---util
|   |           \---util
|   +---unix
|   +---windows
|   +---winprocess
|   \---xml
```

Figure 7 - Java Implementation of the AST

The *lib* directory contains implementation files for the AST. The API for the AST is described in the *src\idl\ow_corba_ast_api.idl* IDL source file.

All implementation files are inherited from the *DeclarationImpl.java* file. This class provides methods to load declarations in the AST from the IR. These methods are *load()* for an IDL3 declaration and *loadAsMapping()* for an IDL2 declaration.

After the OpenCCM compilation process, all classes of the API implementation are generated in the *<used_ORB_name>\classes\org\objectweb\corba\ast* directory.

The *ir3_feed* command line tool of OpenCCM, is used to create instances of the meta-types that correspond to the IDL3 declarations contained in the IDL3 file given as parameter. Before using *ir3_feed*, as explained in the "Getting started with OpenCCM" user guide, the IR3 must be started with the *ir3_start* command. This last process feeds the OpenCCM's IDL3 Repository with the *IFR3_0.idl* file (the OMG IDL file for the OMG IDL3 Repository description) and the *Components.idl* file (the OMG IDL for the CCM's Components module).

2.4 Implementation of the OpenCCM generators

The OpenCCM generators architecture:

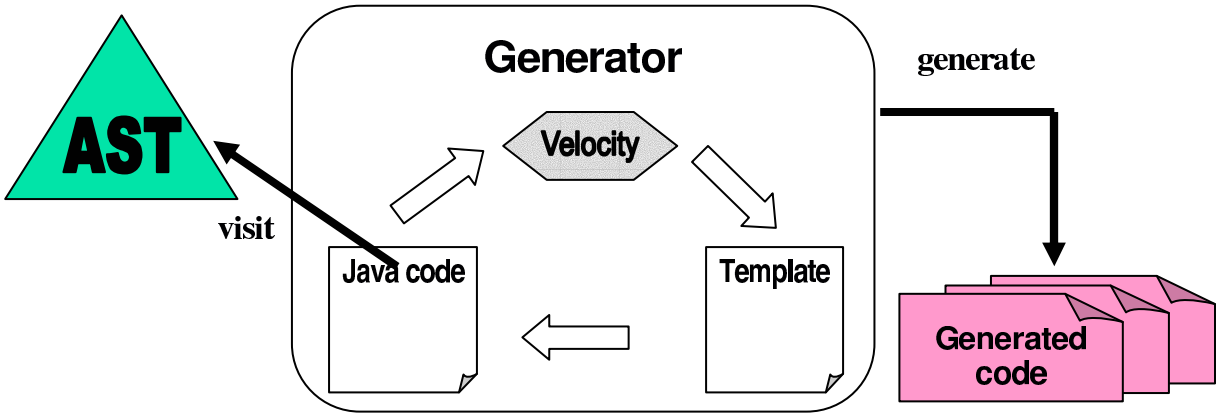

Figure 8 - The OpenCCM Generators Architecture

Using declarations provided by the AST, the generator is based on the Velocity engine (http://jakarta.apache.org/velocity/). This engine uses template files in order to generate code.

The OpenCCM generators' implementations are located in the following packages:

```
+---src
|   +---doxygen
|   +---dtd
|   +---idl
|   +---java
|   |   \---org
|   |       \---objectweb
|   |           +---ccm
|   |           +---corba
|   |           |   +---ast
|   |           |   +---command
|   |           |   +---generator
|   |           |   |   +---cidl
|   |           |   |   +---cif
|   |           |   |   +---common
|   |           |   |   +---idl
|   |           |   |   +---java
|   |           |   |   \---psdl
|   |           |   +---parser
|   |           |   \---util
|   |           \---util
|   +---unix
|   +---windows
|   +---winprocess
|   \---xml
```

Figure 9 - The OpenCCM Generators Implementation

2.4.1 The Basic Generator

Generators are based on the Velocity template engine and the *.vm* template files to generate class Java code. The common template file for all generators is *common.vm* file.

The *common* directory contains:

- *Generator.java* : the Velocity wrapper
- *GeneratorBase.java* : implements methods to visit the AST and methods for generation
- *Indentor.*java: class to indent code

The basic generator is used as a base for other specific generators (IDL, PSDL, CIDL and CIF). For the IDL, PSDL and CIDL generators, the object found in the AST is sent to a Velocity macro associated to this object. For the Java generator, the generator visit the AST and objects are created in the Java AST (which is located in the *org.objectweb.corba.generator.java.ast* package).

```
+---generator
|   +---cidl
|   +---cif
|   +---common
|   |   |   common.vm
|   |   |
|   |   +---api
|   |   |       Generator.java
|   |   |       GeneratorBase.java
|   |   |
|   |   \---lib
|   |           GenerationException.java
|   |           Generator.java
|   |           GeneratorBase.java
|   |           Indentor.java
|   |
|   +---idl
|   +---java
|   \---psdl
```

Figure 10 - The Basic Generator

The next part describes implementation files for specific generators. UML class diagram of generators can be found in annexes.

2.4.2 The OpenCCM OMG IDL Generator

```
+---generator
|   +---cidl
|   +---cif
|   +---common
|   +---idl
|   |   |   idl3.vm
|   |   |
|   |   +---api
|   |   |       IDL2Generator.java
|   |   |       IDL3Generator.java
|   |   |
|   |   \---lib
|   |           IDL2Generator.java
|   |           IDL3Generator.java
|   |           IDLTranslator.java
|   |
|   +---java
|   \---psdl
```

Figure 11 - The OMG IDL3 and IDL2 Generators

The OpenCCM IDL generator allows to generate OMG IDL3 definitions associated to an IR3 object. The Java source files *IDL3Generator.java* (interface in *api* subdirectory and implementation in *lib* subdirectory), and the *idl3.vm* template describe the generation of IDL3 files.

The generation of OMG IDL2 mappings are implemented in *IDL2Generator.java* file.

2.4.3 The OpenCCM PSDL Generator

```
+---generator
|   +---cidl
|   +---cif
|   +---common
|   +---idl
|   +---java
|   \---psdl
|       |   psdl.vm
|       |   psdl2java.vm
|       |
|       +---api
|       |       PSDLGenerator.java
|       |
|       \---lib
|               PSDL2JavaGenerator.java
|               PSDLGenerator.java
|               PSDLTranslator.java
```

Figure 12 - The OpenCCM PSDL Generator Implementation

The OpenCCM Persistent State Definition Language (PSDL) generator allows to generate PSDL definitions describing persistent state of a component, from IDL, PSDL or CIDL files.

PDSLGenerator.java file in *lib* subdirectory implements the PSDL generator interface.

Note that as illustrated in the figure 6, PSDL and CIDL declarations are present only in the AST, contrary to IDL declarations which are automatically registered in the IR once having put in the AST .

2.4.4 The OpenCCM CIDL Generator

```
+---generator
|   +---cidl
|   |   |   cidl.vm
|   |   |
|   |   +---api
|   |   |       CIDLGenerator.java
|   |   |
|   |   \---lib
|   |           CIDLGenerator.java
|   |           CIDLTranslator.java
|   |
|   +---cif
|   +---common
|   +---idl
|   +---java
|   \---psdl
```

Figure 12 - The OpenCCM CIDL Generator Implementation

The OpenCCM CIDL generator allows to generate CIDL definitions describing component implementations, from IDL, CIDL or PSDL files.

2.4.5 The OpenCCM CIF Generator

```
+---generator
|   +---cidl
|   +---cif
|   |   |   cif.vm
|   |   |
|   |   +---api
|   |   |       CIF_IDLGenerator.java
|   |   |       CIF_JavaGenerator.java
|   |   |
|   |   \---lib
|   |           CIF_IDLGenerator.java
|   |           CIF_JavaGenerator.java
|   |
|   +---common
|   +---idl
|   +---java
|   \---psdl
```

Figure 13 - The OpenCCM CIF Generator Implementation

CIF_IDLGenerator.java class in *lib* subdirectory implements the CIF generator interface (component and home executor API). As for others generators, it extends the *GeneratorBase.java* class and generate CIF from declarations of the AST. *CIF_JavaGenerator.java* class then generate classes for CIF implementation (component executor implementation skeletons).

2.5 OpenCCM Production Chain Commands

Java source files of OpenCCM production chain commands are located in *src\java\org\objectweb\corba\command* directory

```
+---src
|   +---doxygen
|   +---dtd
|   +---idl
|   +---java
|   |   \---org
|   |       \---objectweb
|   |           +---ccm
|   |           +---corba
|   |           |   +---ast
|   |           |   +---command
|   |           |   |   +---api
|   |           |   |   \---lib
|   |           |   +---generator
|   |           |   +---parser
|   |           |   \---util
|   |           \---util
|   +---unix
|   +---windows
|   +---winprocess
|   \---xml
```

Figure 14 - OpenCCM Production Chain Commands

The *lib* directory contains Java implementation files of OpenCCM commands (IDL3, IDL2, CIDL and PSDL generations):

```
+---command
|   +---api
|   \---lib
|           ApplicationBase.java
|           ApplicationServerBase.java
|           CIDLtoCIDL.java
|           CIDLtoCIF.java
|           CommandOnASTBase.java
|           CommandOnIR3Base.java
|           CompilerBase.java
|           CompilerGeneratorBase.java
|           GeneratorBase.java
|           IDL3Check.java
|           IR3Destroy.java
|           IR3Feed.java
|           IR3toIDL2.java
|           IR3toIDL3.java
|           OptionMultipleArguments.java
|           PSDLtoPSDL.java
```

Figure 15 - Java Implementation of OpenCCM Commands

The *api* directory contains interfaces of the main OpenCCM command scripts implemented in previous *lib* directory, for IDL3, IDL2, CIDL, PSDL and CIF generations:

```
+---command
|   +---api
|   |       Application.java
|   |       ApplicationServer.java
|   |       CIDLtoCIDL.java
|   |       CIDLtoCIF.java
|   |       CommandOnAST.java
|   |       CommandOnIR3.java
|   |       Compiler.java
|   |       Generator.java
|   |       IDL3Check.java
|   |       IR3Destroy.java
|   |       IR3Feed.java
|   |       IR3toIDL2.java
|   |       IR3toIDL3.java
|   |       PSDLtoPSDL.java
|   |
|   \---lib
```

Figure 16 - Interfaces for OpenCCM Command

The following figure describes dependancies between OpenCCM command classes:

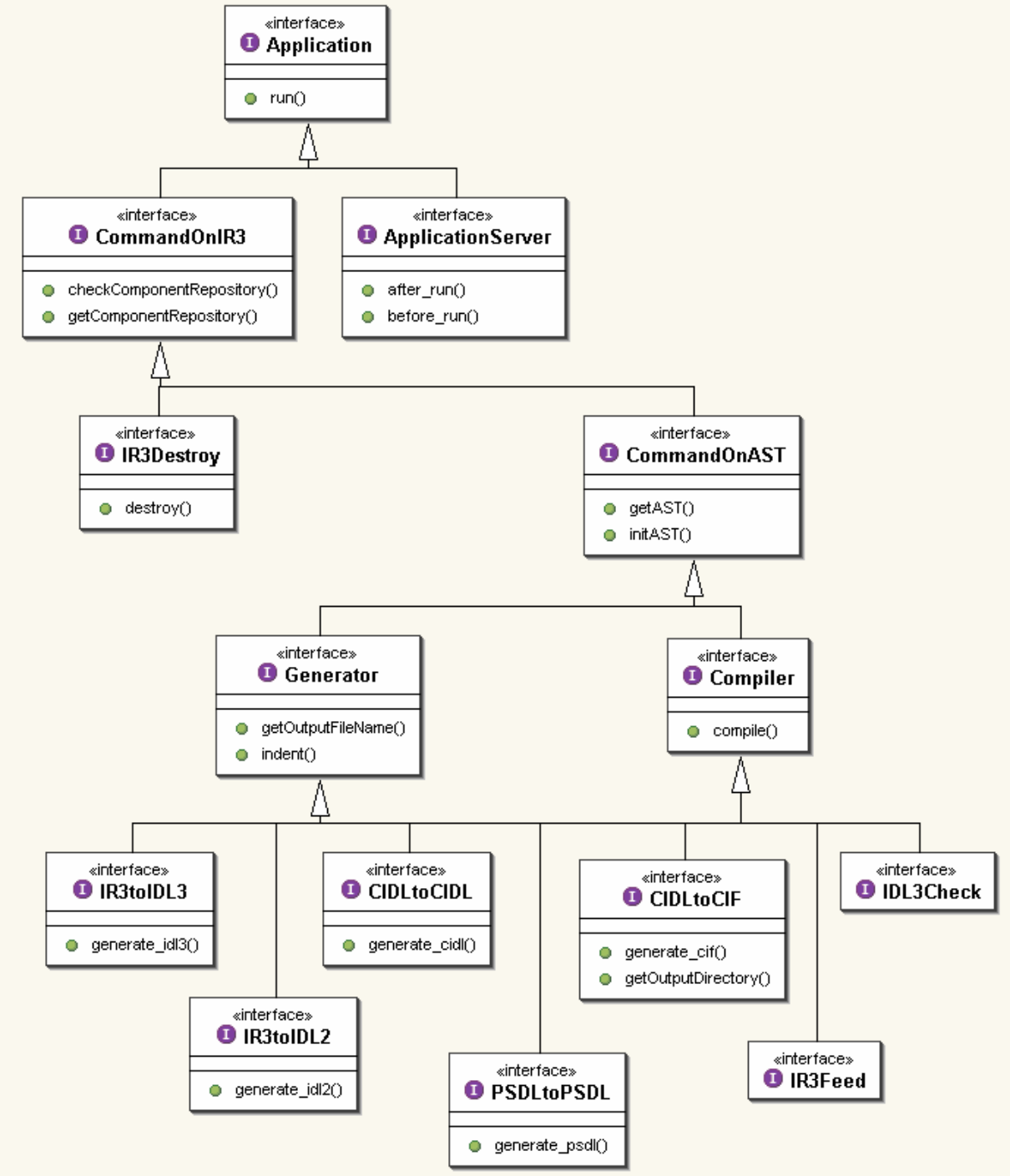

Figure 17 - OpenCCM Command Classes Diagram

In the following paragraphs, we detail each command of OpenCCM production chain's generators.

2.5.1 About OpenCCM Command Scripts

There are two ways to run an OpenCCM command:

- Directly executing the *.bat* or *.sh* file depending on the OS platform (Windows or Unix), which start the Java virtual machine with classpath option for externals Java archives dependencies,
- Using the Java launcher to execute scripts. This launcher uses XML configuration files to start the command. Theses XML files describes which Java archive files or other XML files are necessary to execute the command

XML configuration files are based on the following DTD:

```
<!ELEMENT launcherConfig

    ( include    | run    | classpath    | arguments    | properties    )* >

<!ELEMENT include EMPTY >

<!ATTLIST include    url CDATA #REQUIRED>


<!ELEMENT run EMPTY >

<!ATTLIST run  id CDATA #REQUIRED

               mainclassname CDATA #REQUIRED

               classpath IDREF #IMPLIED

               arguments IDREF #IMPLIED

               properties IDREF #IMPLIED>


<!ELEMENT classpath    (path)*>

<!ATTLIST classpath    id ID #REQUIRED>


<!ELEMENT path EMPTY >

<!ATTLIST path   url CDATA #IMPLIED

                 classpath IDREF #IMPLIED>


<!ELEMENT arguments    (argument)*>

<!ATTLIST arguments    id ID #REQUIRED>


<!ELEMENT argument EMPTY >

<!ATTLIST argument  value CDATA #IMPLIED

          arguments IDREF #IMPLIED>
```

```
<!ELEMENT properties    (property)*>

<!ATTLIST properties    id ID #REQUIRED>

<!ELEMENT property EMPTY >

<!ATTLIST property    name CDATA #IMPLIED

                      value CDATA #IMPLIED

                      properties IDREF #IMPLIED>
```

Figure 18 - The Launcher DTD

The main elements needed in an XML configuration file are : *include*, *run*, *classpath*, *arguments* and *properties*. Those elements are uniquely identified by an ID. This ID is the same for all elements of all included files.

The *include* element allows to include other XML configuration files. The *run* element indicate which is the main class to run. *classpath* attribut of this run element give it Java archive to add to the classpath. Through arguments and properties elements, the launcher can replace a "*@tag@* like" by its Java associated property.

Here is two examples of XML configuration files, used by many OpenCCM command:

ORB.xml

```
<launcherconfig>

    <classpath id="orb">

        <path url="file:C:/Orbacus/OB-4.1.0/lib/OB.jar" />

        <path url="file:C:/Orbacus/OB-4.1.0/lib/OBNaming.jar" />

        <path url="file:C:/Orbacus/OB-4.1.0/lib/OBUtil.jar" />

   </classpath>

</launcherconfig>
```

ProductionChain.xml

```
<launcherConfig>

 <include url="@LAUNCHER_XML_DIR@/ORB.xml" />

     <classpath id="cp_production_chain">

           <path url="file:@LIB_DIR@/OpenCCM.jar" />

           <path classpath="orb" />

     </classpath>
```

```
 <arguments id="arg_production_chain">

  < <argument value="-ORBInitRef
InterfaceRepository=file:@OpenCCM_CONFIG_DIR@@file.separator@IR3.IOR" />

 </arguments>

</launcherConfig>
```

Figure 19 - XML Configuration Files for ORB and Production Chain

2.5.2 Starting the OpenCCM's OMG IDL3 Repository (IR3)

Dependencies and entry point for *ir3_start* command are so:

- OpenCCM Java archive file (OpenCCM.jar)
- The XML configuration file for the installed ORB (ORB.xml, listing Java archive for the used ORB)
- The org.objectweb.corba.command.lib.IR3Install class

The *ir3_start* command runs IR3Install class which is located in *\src\java\org\objectweb\corba\command* directory

```
<launcherConfig>

    <include url="@LAUNCHER_XML_DIR@/ProductionChain.xml" />

    <run id="default"

         mainclassname="org.objectweb.ccm.scripts.IR3Install"

         classpath="cp_productionchain"

    />

</launcherConfig>
```

Figure 20 – The XML Configuration File of OpenCCM's IR3Install

The *lib* directory contains Java implementation files for the *ir3_start* command and *api* directory the APIs.

2.5.3 Feeding the OpenCCM IR3 with an OMG IDL3 file

The *ir3_feed* command script compiles the IDL3 file and feed the OpenCCM's IR3. As for the *ir3_start* command (and others), Java source files for *ir3_feed* are located in *\src\java\org\objectweb\corba\command* directory (the *lib* directory for the implementations and the *api* directory for the interfaces).

```
<!--  Elements required to run the IR3Feed script.                    -->

<launcherConfig>

    <include url="@LAUNCHER_XML_DIR@/ProductionChain.xml" />

    <arguments id="arg_ir3_feed">

        <argument value="-I@IDL_DIR@" />

    </arguments>

    <run id="default"

         mainclassname="org.objectweb.corba.command.lib.IR3Feed"

         classpath="cp_productionchain"

         arguments="arg_ir3_feed"

    />

</launcherConfig>
```

Figure 21 - The XML Configuration File the OpenCCM's IR3Feed

The *ir3_feed* command run the *org.objectweb.corba.command.lib.IR3Feed* class, and dependencies in XML configuration file are *OpenCCM.jar* archive (build after compilation and installation of OpenCCM platform in *build\lib* directory) , ORB.xml file and a *argument* XML element to include necessary IDL files (located in *\build\idl* directory generated after compilation process).

2.5.4 Generator for equivalent OMG IDL 2.4 mappings from an OMG IDL3 file

The *ir3_idl2* command generates OMG IDL2.4 mappings associated to an OpenCCM's IR3 object. It uses the *org.objectweb.corba.command.lib.IR3toIDL2* class to run the generator.

Java source files for *ir3_idl2* command are *IR3toIDL2.java* (implementation, in *lib* subdirectory) and *IR3toIDL2Operations.java* (the interface, in *api* subdirectory).

The generator uses the Velocity tool to generate code, from a template file, dependencies for *ir3_idl2* command are:

- The *OpenCCM.jar* archive
- The *ORB.xml* XML configuration file
- A *Velocity.xml* XML configuration file that contains all necessary Java archives to run Velocity

```
<!--  Elements required to run the IR3toIDL2 command.                    -->

<launcherConfig>

    <include url="@LAUNCHER_XML_DIR@/Generator.xml" />

    <run id="default"

         mainclassname="org.objectweb.corba.command.lib.IR3toIDL2"

         classpath="cp_generator"

         properties="prop_generator"

    />

</launcherConfig>
```

Figure 22 - The XML Configuration File the OpenCCM's IR3toIDL2 generator

2.5.5 OpenCCM IDL3 Generator Command

The *ir3_idl3* command runs the *org.objectweb.corba.command.lib.IR3toIDL3* class and allows to generate IDL3 description for a declaration of the repository. This class uses the IDL3 generator in *\org\objectweb\corba\generator\idl* directory.

Required elements for this script are the same as in the IDL2 generator (*OpenCCM.jar* archive, *ORB.xml* and *Velocity.xml* configuration files).

2.5.6 OpenCCM PSDL Generator Command

This command run *org.objectweb.corba.command.lib.PSDLtoPSDL* class and allow to generate PSDL from a PSDL file. This class uses the PSDL generator in *org\objectweb\corba\generator\psdl* directory.

Required elements for this command are the same as IDL2 generator (*OpenCCM.jar* archive, *ORB.xml* and *Velocity.xml* configuration files).

2.5.7 OpenCCM CIDL Generator Command

CIDL to CIDL command run *org.objectweb.corba.command.lib.CIDLtoCIDL* class and allows to generate CIDL from a CIDL file. This class uses the CIDL generator in *org\objectweb\corba\generator\cidl* directory. It also extends the PSDLtoPSDL class and implements *CIDLtoCIDLOperations* class in *api* subdirectory.

Required elements for this script are the same as IDL2 generator's ones (*OpenCCM.jar* archive, *ORB.xml* and *Velocity.xml* configuration files).

2.5.8 CIF and Java Component Segmented Implementation Skeletons Generator

org.objectweb.corba.command.lib.CIDLtoCIF class allows to generate Java skeletons implementation of homes, components and segments. Indeed, the Component Implementation Framework (CIF) programming model, through the Component Implementation Definition Language (CIDL), is used to:

- describe component composition and how components should be implemented,
- generate executor skeletons which provide segmentation of component executors and implementation of callback operations
- manage component's persistent state with OMG Persistent Definition Language (PSDL)

The next figure depicts dependencies between theses different languages:

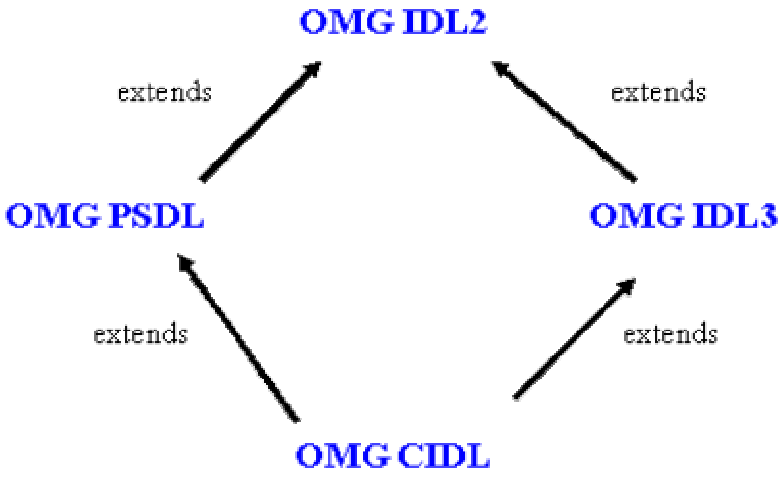

Figure 23 – OMG IDL, PSDL and CIDL dependencies

CIDLtoCIF class extends the *IR3toIDL2* class and do the following:

- runs the CIF IDL generator (*CIF_IDLGenerator* class in *org.objectweb.corba.generator.cif.lib* package),
- creates an IDL2 repository and launch the IDL2 generator,
- generates CIF implementation with *CIF_JavaGenerator*,
- and deletes CIF delarations in the IR.

Using these generated Java implementation skeletons for homes, compositions and segments, developpers only have to concentrate on business code to write component implementations.

2.5.9 Java Container Classes Generator

The Java container generator (*ir3_java command tool*) is used to generate container specific classes, such as the component extended skeletons or interceptors. As described in the CCM specification, non-business operations (home implicit operations, connection operations, etc) are implemented in the generated extended skeletons.

The *ir3_java* command runs the *org.objectweb.ccm.scripts.IR3toJava* class which generates Java OpenCCM skeletons. This class extends *org.objectweb.ccm.CORBA.Application* and do the following things when launched:

- Obtains the reference of the Interface Repository,
- starts an AST binded to the IR,
- Generate Java skeletons using *org.objectweb.ccm.visitorIDL3.java.IDL3_JavaSkeleton* class.

2.5.10 Java Component Implementation Templates Generator

The *ir3_jimpl* command runs the *org.objectweb.ccm.scripts.IR3toJavaImpl* class which generates Java component monolithic implementation templates on which developpers can starts to write their business code in order to implement components and homes of the application. This class extends the *org.objectweb.ccm.CORBA.Application* class and do the following things when launched:

- obtains the IR,
- starts an AST, and
- uses the “org.objectweb.ccm.visitorIDL3.java.IDL3_JavaImpl” class to generate Java template classes.

2.5.11 XMI 1.1 Generator

The *ir3_xmi* command run the *org.objectweb.ccm.scripts.IR3toXMI* class which generate XMI 1.1 UML documents. The XMI generator package is composed of the following classes:

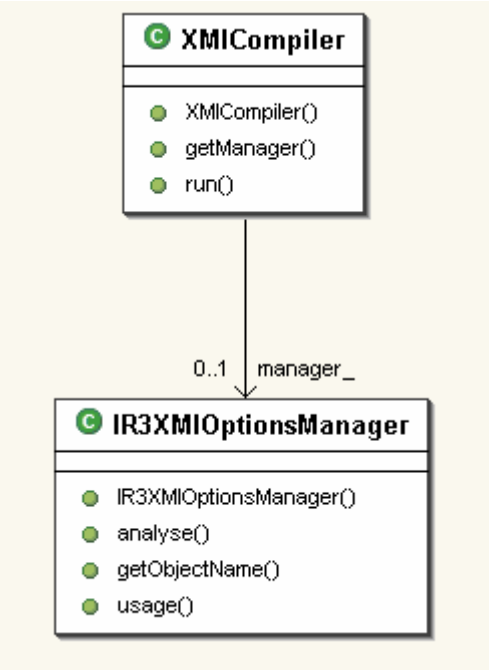

Figure 24 - The XMI Generator Package

To use this command, the OMG IDL3 interface repository must be started with the *ir3_start* command. Here is an example of use of the *ir3_xmi* command to generate a XMI 1.1 UML file.

```
C:\ORB3\OpenCCM-0.6\demo\demo1>ccm_install
The OpenCCM Platform will be installed.
Creating the C:\ORB3\OpenCCM-0.6\ORBacus-4.1\OpenCCM_CONFIG_DIR directory.
Creating the C:\ORB3\OpenCCM-0.6\ORBacus-4.1\OpenCCM_CONFIG_DIR\ComponentServers
directory.
The OpenCCM Platform is installed.

C:\ORB3\OpenCCM-0.6\demo\demo1>ir3_start
The OpenCCM's OMG IDL3 Repository will be started.
Launching the OpenCCM's IR3.
Feeding the OpenCCM's IR3 with the IFR_3_0.idl file.
ir3_feed 0.6: Reading from file C:\ORB3\OpenCCM-0.6\ORBacus-4.1\idl\IFR_3_0.idl...
ir3_feed 0.6: Preprocessing file C:\ORB3\OpenCCM-0.6\ORBacus-4.1\idl\IFR_3_0.idl...
ir3_feed 0.6: File C:\ORB3\OpenCCM-0.6\ORBacus-4.1\idl\IFR_3_0.idl preprocessed.
ir3_feed 0.6: Compiling C:\ORB3\OpenCCM-0.6\ORBacus-4.1\idl\IFR_3_0.idl file...
ir3_feed 0.6: Compilation completed: 0 warning.
Feeding the OpenCCM's IR3 with the Components.idl file.
ir3_feed 0.6: Reading from file C:\ORB3\OpenCCM-0.6\ORBacus-
4.1\idl\Components.idl...
ir3_feed 0.6: Preprocessing file C:\ORB3\OpenCCM-0.6\ORBacus-
4.1\idl\Components.idl...
ir3_feed 0.6: File C:\ORB3\OpenCCM-0.6\ORBacus-4.1\idl\Components.idl preprocessed.
ir3_feed 0.6: Compiling C:\ORB3\OpenCCM-0.6\ORBacus-4.1\idl\Components.idl file...
ir3_feed 0.6: Compilation completed: 0 warning.
The OpenCCM's OMG IDL3 Repository is started.
C:\ORB3\OpenCCM-0.6\demo\demo1>ir3_feed demo1.idl3
```

```
ir3_feed 0.6: Reading from file demo1.idl3...
ir3_feed 0.6: Preprocessing file demo1.idl3...
ir3_feed 0.6: File demo1.idl3 preprocessed.
ir3_feed 0.6: Compiling demo1.idl3 file...
ir3_feed 0.6: Compilation completed: 0 warning.

C:\ORB3\OpenCCM-0.6\demo\demo1>ir3_xmi -o demo1.xmi demo1
Property file null\jidlscript.properties not found: exiting
```

Figure 25 - The XMI 1.1 UML Generator Command

By default, the *ir3_xmi* generate XMI file on standard output. Use *–o* option to redirect output to a specific xmi file.

2.6 IDL, XML, DTD and Command Scripts Source Directories

2.6.1 OpenCCM IDL Sources

The *src\idl* directory contains the following files:

```
+---src
|   +---idl
|   |       Components.idl
|   |       CosTransactions.idl
|   |       Deployment.idl
|   |       IFR_3_0.idl
|   |       OpenCCM.idl
|   |       ow_corba_ast_api.idl
|   |       ow_corba_parser_api.idl
|   |       Transaction_Plugins.idl
```

Figure 26 - OpenCCM IDL Sources

The *Components.idl* file contains the IDL2 specification of:

- The base client-side interfaces supported by any component and home (e.g. CCMObject and CCMHome) references,
- The base server-side interfaces supported by any component and home implementation (named executors in CCM), and
- The container related interfaces, *i.e.* callbacks interfaces (*e.g.* SessionComponent) and container internal interfaces (*e.g.* SessionContext)

All these interfaces are described in the chapters 1 (“Component Model”), 3.3 (“CCM Implementation Framework”, “Language Mapping” section) and 4 (“The Container Programming Model”) of the CCM specification. Note that this file contains some changes from the specification. Note also that this file is incomplete with respect to the specification.

OpenCCM.idl file describe OpenCCM specific API, such as:

- *ccm* module: interface to shutdown OpenCCM servers
- *ccm::Deployment* specifications: obtain component servers
- interfaces for the implementation of features during the deployment process
- interfaces for containers
- etc.

org_objectweb_corba_ast_lib.idl is the OMG IDL for the OpenCCM Abstract Syntax Tree. *IFR_3_0.idl* file provides definitions for the OMG IDL3 Repository.

Deployment.idl file is the OMG IDL for the CCM *Components::Deployment* module and provides interfaces such as: ServerActivator, ComponentServer and Container.

2.6.2 OpenCCM Unix and Windows Command Scripts Sources

src\unix and *src\windows* directories contain Operating System (OS) specific command scripts for OpenCCM production chain and execution infrastructure. Depending on the user's platform, these scripts are used as source files during compilation and installation of OpenCCM.

```
+---src
|   +---doxygen
|   +---dtd
|   +---idl
|   +---java
|   +---unix
|   |       ccm_deinstall
|   |       ccm_deploy
|   |       ccm_install
|   |       ccm_installed
|   |       cidl
|   |       cidl_cif
|   |       idl3_check
|   |       ir3_destroy
|   |       ir3_feed
|   |       ir3_idl2
|   |       ir3_idl3
|   |       ir3_java
|   |       ir3_jimpl
|   |       ir3_start
|   |       ir3_started
|   |       ir3_stop
|   |       ir3_xmi
|   |       jcs_start
|   |       jcs_stop
|   |       ns_ior
|   |       ns_started
|   |       ns_stop
|   |       old_ir3_feed
|   |       old_ir3_idl2
|   |       old_ir3_idl3
|   |       ots_ior
|   |       ots_stop
|   |       psdl
```

```
|   |       shutdown.sh
|   |
|   +---windows
|   +---winprocess
|   \---xml
```

Figure 27 - OpenCCM Unix and Windows Command Scripts Sources

2.6.3 OpenCCM DTD Sources

The *src\dtd* directory contains three subdirectories: *ccm, launcher* and *xmi*. The *ccm* subdirectory contains the following files:

```
+---src
|   +---dtd
|   |   +---ccm
|   |   |       componentassembly.dtd
|   |   |       corbacomponent.dtd
|   |   |       properties.dtd
|   |   |       softpkg.dtd
|   |   |
|   |   +---launcher
|   |   |       launcher.dtd
|   |   |
|   |   \---xmi
|   |           XMI_1_1_UML.dtd
|   |
```

Figure 28 – OpenCCM DTD Files

i) DTD for the Deployment Tool

- *componentassembly.dtd* is the DTD for the Component Assembly Descriptor *<assembly>.cad*
- *corbacomponent.dtd* is the DTD for the Corba Component Descriptor *<component>.ccd*
- *properties.dtd* is the DTD for the Property File Descriptor *<file>.cpf*
- *softpkg.dtd* is the DTD for the Software Package Descriptor *<package>.csd*

The *xmi* subdirectory contains the *XMI_1_1_UML.dtd* DTD for generated XMI files.

ii) The Launcher DTD

The launcher.dtd is the DTD for the launcher tool which allows to run OpenCCM command of the production or execution chain. See **§ 2.5.1 About OpenCCM commands** and the *Figure 18 - The Launcher DTD* for more details.

2.6.4 OpenCCM XML Configuration File Sources

The *xmlNauncher* subdirectory contains XML configuration files allowing to execute classes of commands of OpenCCM production chain. These XML files describe which Java archive files, classes, Java properties, arguments or other XML files are necessary to execute the related command, like explained in the paragraph 2.5.1.

```
|   \---xml
|       +---launcher
|       |       CIDLtoCIDL.xml
|       |       CIDLtoCIF.xml
|       |       Generator.xml
|       |       IDL3Check.xml
|       |       IR3Destroy.xml
|       |       IR3Feed.xml
|       |       IR3Install.xml
|       |       IR3toIDL2.xml
|       |       IR3toIDL3.xml
|       |       ProductionChain.xml
|       |       PSDLtoPSDL.xml
|       |       Velocity.xml
|       |
|       \---zeusx
|               componentassembly.zeusx
|
```

Figure 29 - XML Configuration File Sources

3 OPENCCM RUNTIME

3.1 The OpenCCM Components Runtime

Through the *org.objectweb.ccm.Components* package, OpenCCM platform provides implementations for CORBA components and homes base interfaces. Java source files of this package are:

```
|   |   \org\objectweb\ccm
|   |           |   +---Components
|   |           |   |       CCMHomeBase.java
|   |           |   |       CCMHomeImpl.java
|   |           |   |       CCMHomeWithPKImpl.java
|   |           |   |       CCMObjectImpl.java
|   |           |   |       ComponentPortDescriptionFactory.java
|   |           |   |       ComponentPortDescriptionImpl.java
|   |           |   |       ConfiguratorImpl.java
|   |           |   |       ConfigValueFactory.java
|   |           |   |       ConfigValueImpl.java
|   |           |   |       ConnectionDescriptionFactory.java
|   |           |   |       ConnectionDescriptionImpl.java
|   |           |   |       ConsumerDescriptionFactory.java
|   |           |   |       ConsumerDescriptionImpl.java
|   |           |   |       ConsumesInfo.java
|   |           |   |       CookieFactory.java
|   |           |   |       CookieImpl.java
|   |           |   |       EmitsInfo.java
|   |           |   |       EmitterDescriptionFactory.java
|   |           |   |       EmitterDescriptionImpl.java
|   |           |   |       FacetDescriptionFactory.java
|   |           |   |       FacetDescriptionImpl.java
|   |           |   |       HomeManagerImpl.java
|   |           |   |       MonolithicWrapperBase.java
|   |           |   |       PortDescriptionImpl.java
|   |           |   |       PortInfo.java
|   |           |   |       ProvidesInfo.java
|   |           |   |       PublisherDescriptionFactory.java
|   |           |   |       PublisherDescriptionImpl.java
|   |           |   |       PublishesInfo.java
|   |           |   |       ReceptacleDescriptionFactory.java
|   |           |   |       ReceptacleDescriptionImpl.java
|   |           |   |       Runtime.java
|   |           |   |       StandardConfiguratorImpl.java
|   |           |   |       SubscriberDescriptionFactory.java
|   |           |   |       SubscriberDescriptionImpl.java
|   |           |   |       UsesInfo.java
```

Figure 30 - The org.objectweb.ccm.Components package

- *CCMHomeBase.java* interface and *CCMHomeImpl.java* classes implement all generic operations for CORBA Component Model homes. Home is a new CORBA meta-type that manages a unique component type (factory design pattern). It is instantiated at deployment time.
- *CCMObjectImpl.java* class implements all generic operations for CCM components. Component is also a new CORBA meta-type and is an extension of classical interfaces. A component can provide the following ports: facets, receptacles, event sources and event sinks, and support classical CORBA attributes for its configuration.

The Java container skeletons generated by the OpenCCM production chain inherit from theses *org.objectweb.ccm.runtime.Components* implemention classes.

The next figure shows the class diagram for *CCMObjectImpl* class:

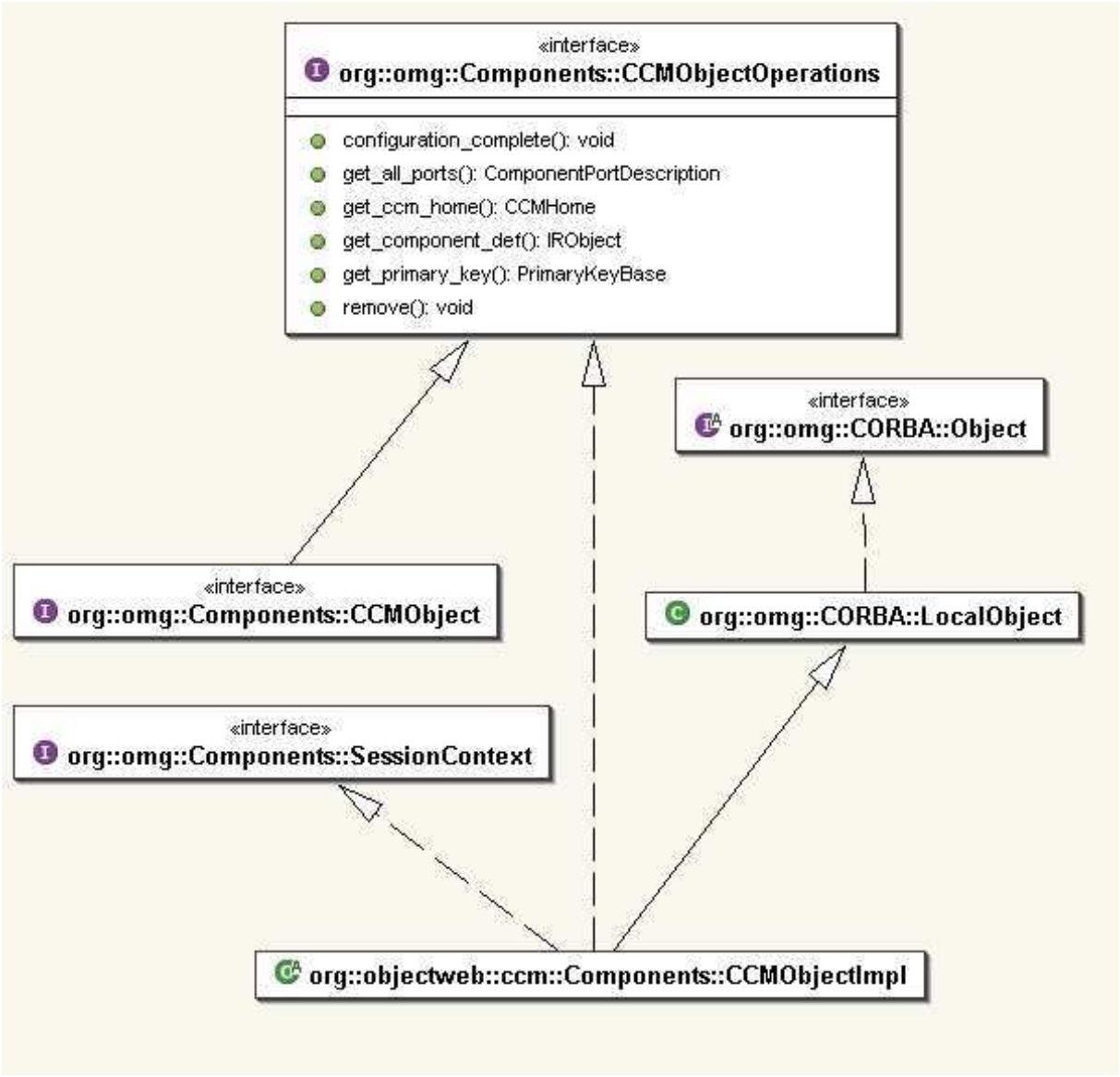

Figure 31 - The CCMObjectImpl class diagram of OpenCCM Components Runtime

As shown in this figure, the *CCMObjectImpl* class implements the base *Components::CCMObject* interface through the *Components::CCMObjectOperations* interface and also the *Components::SessionContext* interface. In this way, the *CCMObjectImpl* class serves as a base implementation for the components extended skeletons generated by OpenCCM.

3.2 The OpenCCM Containers Runtime

The OpenCCM container runtime is implemented by the *org.objectweb.ccm.runtime.Containers* package. The container provides simplified interfaces for CORBA services (security, transactions, persistence and notification). Container object is created by component servers and host executors. CCMHome objects are also installed by containers, and component instances are created and managed at runtime by its container.

The next figure shows the container architecture as specified by the OMG:

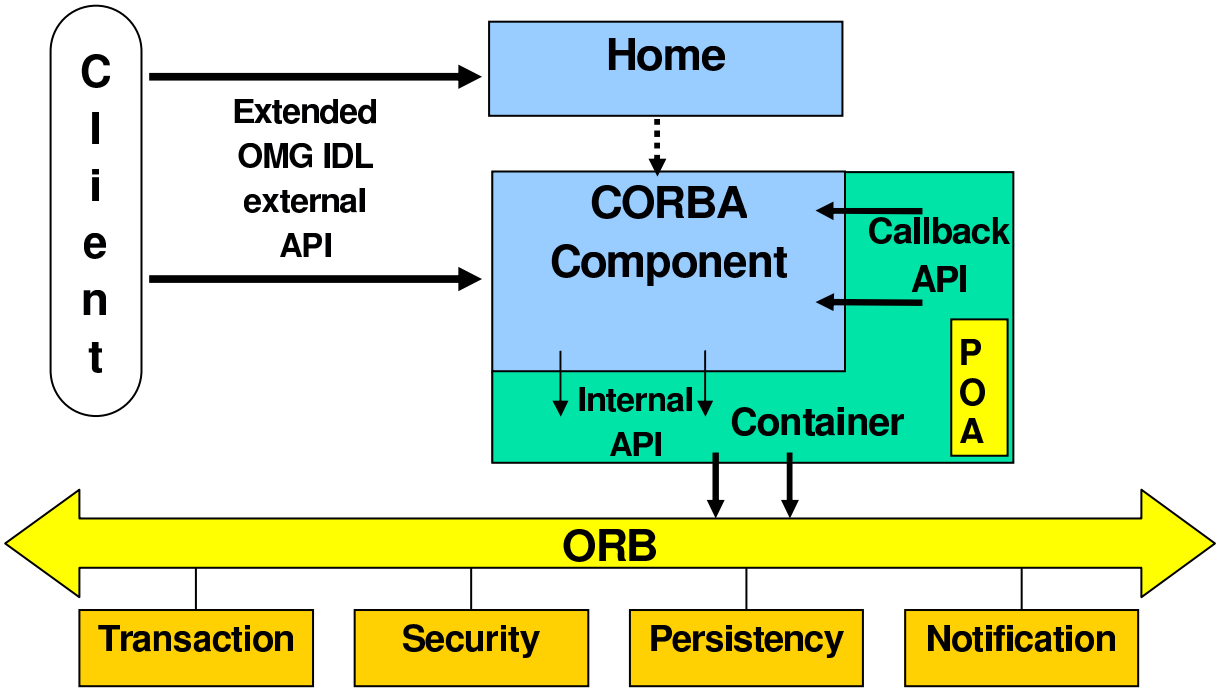

Figure 32 - The Container Architecture

The *org.objectweb.ccm.runtime.Containers* package contains the next Java source files:

```
|   |   \---org
|   |       \---objectweb
|   |           +---ccm
|   |           |   +---Containers
|   |           |   |   |   CallContextBase.java
|   |           |   |   |   ComponentCallContext.java
|   |           |   |   |   ComponentExecutor.java
|   |           |   |   |   ComponentServant.java
|   |           |   |   |   ComponentServantImpl.java
|   |           |   |   |   HomeCallContext.java
|   |           |   |   |   HomeExecutor.java
|   |           |   |   |   HomeExecutorBase.java
|   |           |   |   |   HomeExecutorWithPK.java
|   |           |   |   |   HomeServant.java
|   |           |   |   |   HomeServantImpl.java
|   |           |   |   |   Interceptor.java
|   |           |   |   |   NativePolicy.java
|   |           |   |   |   OperationCallContext.java
|   |           |   |   |   PCAImpl.java
|   |           |   |   |   PropertyImpl.java
|   |           |   |   |   PropertySetImpl.java
|   |           |   |   |   RootPCAImpl.java
|   |           |   |   |   ServantLocatorImpl.java
|   |           |   |   |   StringPropertyImpl.java
|   |           |   |   |
|   |           |   |   \---Plugins
|   |           |   |           EmptyConfiguration.java
|   |           |   |           EmptyConfigurationHome.java
|   |           |   |           EmptyController.java
|   |           |   |           EmptyControllerHome.java
|   |           |   |           EmptyCoordinator.java
|   |           |   |           EmptyCoordinatorHome.java
|   |           |   |           ListCoordinator.java
|   |           |   |           ListCoordinatorHome.java
|   |           |   |           PortSpecificConfiguration.java
|   |           |   |           PortSpecificConfigurationHome.java
|   |           |   |           SingleCallController.java
|   |           |   |           SingleCallControllerHome.java
|   |           |   |           TraceController.java
|   |           |   |           TraceControllerHome.java
|   |           |   |
```

Figure 33 - The org.objectweb.ccm.runtime.Containers Package

The implementation of the container in OpenCCM follows the classical POA approach defined in CORBA. That is:

- A component-oriented POA is defined in the *OpenCCM.idl* file (PCA local interface) and implemented by the PCAImpl class. Note that the *RootPCAImpl* class is the equivalent of the RootPOA in CORBA. Here, it used to manage home instances. The PCA is responsible for the lifecycle management of component servants as well as for the creation/destruction of their references. The PCA implementation is based on a servant locator approach.
- Component-oriented servants are also defined as native types in the *OpenCCM.idl* file and as matter of fact specified through Java interfaces in this package (*ComponentServant* and *HomeServant* interfaces). The goal of these servants is to provide an abstraction of the component implementation (executor) to the PCA.
- Container-oriented executors are not defined as IDL types and serves as an abstraction of the standard CCM executors for the container. Their goal is to provide some additional informations about the component or the home instances to the container. Typically, this information consists of type information (e.g. repository ids) and provide operations to create runtime objects needed by the container (typed interceptors, executor wrappers, etc)

Moreover, the implementation of the configuration mechanism valuetypes is also located in this package.

3.3 The OpenCCM CIF Runtime

The *org.objectweb.ccm.runtime.cif* package contains implementation classes for generated component executor implementation skeletons. These classes are the following:

```
|   |   \---org
|   |       \---objectweb
|   |           +---ccm
|   |       |   |   +---runtime
|   |       |   |   |   \---cif
|   |       |   |   |       +---api
|   |       |   |   |       |       ComponentSegment.java
|   |       |   |   |       |       Segment.java
|   |       |   |   |       |       SegmentHome.java
|   |       |   |   |       |
|   |       |   |   |       \---lib
|   |       |   |   |               ComponentSegmentBase.java
|   |       |   |   |               HomeBase.java
|   |       |   |   |               SessionComponentSegmentBase.java
```

Figure 34 - The OpenCCM CIF Runtime Package

The UML class diagram of these runtimes files looks like:

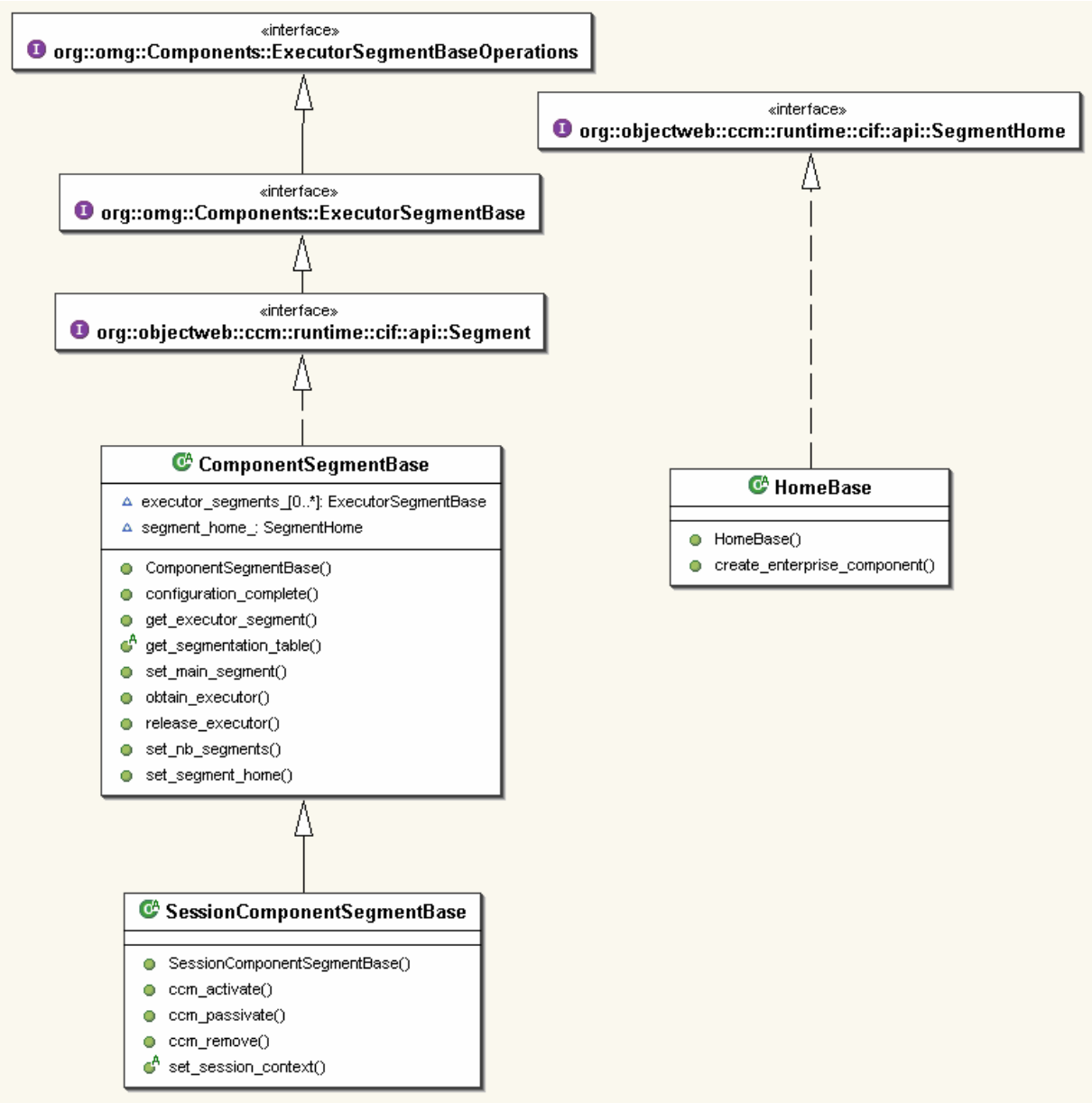

Figure 35 - Class Diagram of the CIF Runtime

The *ComponentSegmentBase* class in *lib* subdirectory implements main operations of the component executor. For a component of "session" type, the *SessionComponentSegmentBase* class is used.

The generated OpenCCM component executor implementation skeletons inherit from these runtimes files, as the following figure details it, in the case of a simple *Client* component (see § 3.4 for more details):

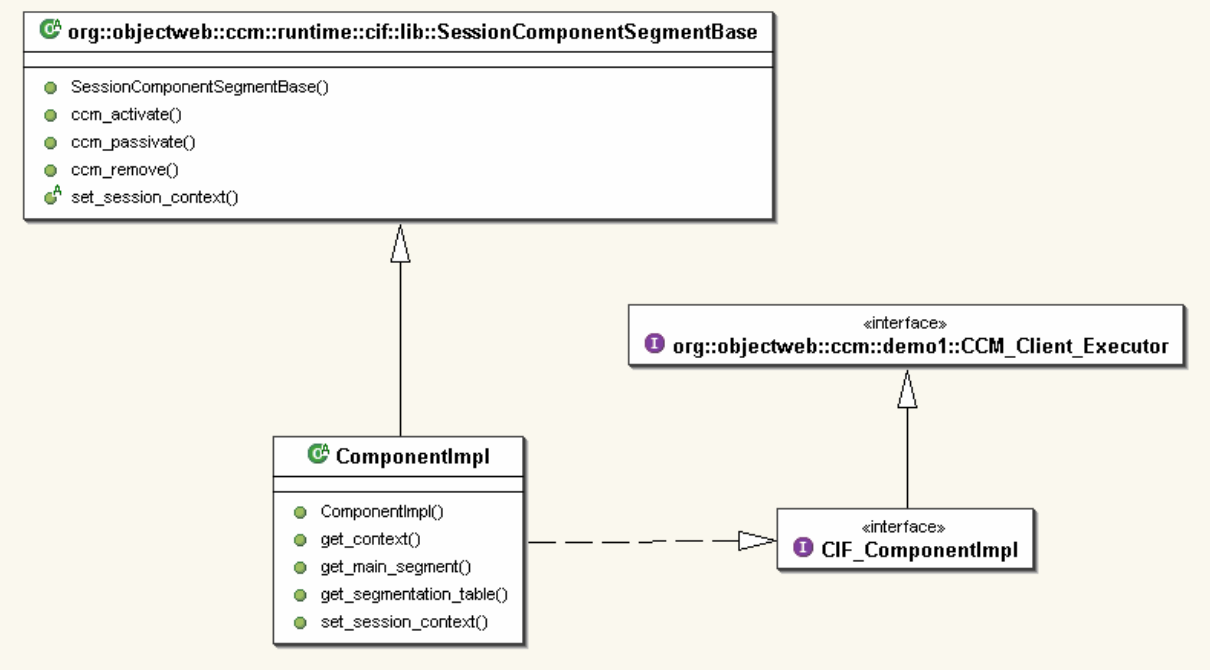

Figure 36 - CIF Runtime and Component Executor Skeletons Dependencies

3.4 A CCM Application Running, How Does it work ?

Now, we are going to see in this part, with a concrete simple example, how components are executed and implemented by the OpenCCM platform, according to the OMG CORBA Component Model specification, and to depandencies between generated classes.

This example is a simple client / server application showing a *Client* component connected by a receptacle to a facet provided by *Server* component (this example is the *demo1* example in OpenCCM distribution).

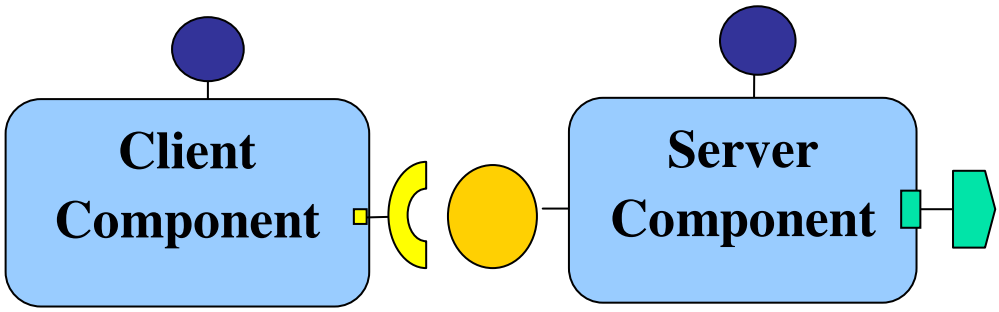

Figure 37 - Our Simple CCM Client / Server Example

3.4.1 The Generated Java Extended Skeletons and Interceptors

OpenCCM generated Java extended skeletons and interceptors for each component are :

```
\---org
    \---objectweb
        \---ccm
            \---demo1
                |   ClientCCM.java
                |   ClientHomeCCM.java
                |   ClientHomeSkeletonInterceptor.java
                |   ClientHomeStubInterceptor.java
                |   ClientMonolithicWrapper.java
                |   ClientSkeletonInterceptor.java
                |   DisplaySkeletonInterceptor.java
                |   DisplayStubInterceptor.java
                |   ServerCCM.java
                |   ServerHomeCCM.java
                |   ServerHomeSkeletonInterceptor.java
                |   ServerHomeStubInterceptor.java
                |   ServerMonolithicWrapper.java
                |   ServerSkeletonInterceptor.java
                |
                +---ClientSessionComposition
                |       ComponentImpl.java
                |       HomeImpl.java
                |
                \---ServerSessionComposition
                        ComponentImpl.java
                        HomeImpl.java
```

Figure 38 - OpenCCM Generated Java Skeletons for Client and Server Components

For each component definition, an extended skeleton is generated in a class named <*component_name*>*CCM*. This skeleton inherits from the *CCMObjectImpl* base class and implements:

- Connection related operations coming from the client-side IDL3 to IDL2 mapping, that is the <*component_name*> IDL2 interface operations,
- Context related operations coming from the server-side IDL3 to IDL2 mapping, that is the *CCM_*<*component_name*>*_Context* local interface operations.

For each home definition, an extended skeleton is also generated in a class named <*home_name*>*CCM*. This skeleton inherits from the *CCMHomeImpl* base class for home without primary key and from the *CCMHomeWithPKImpl* class for home with a primary key. In both cases, this class implements:

- All operations coming from the client-side IDL3 to IDL2 mapping, *i.e.* the <*home_name*> IDL2 interface operations. Note that the operations of the implicit

interface (<*home_name*>*Implicit*) are fully implemented whereas operations of the explicit interface (<*home_name*>*Explicit*) are delegated to the executor.

Moreover, typed interceptors are also generated for both components and homes. These interceptors wrap the extended skeletons at runtime and constitute the integration point for call coordinators and call controllers. Interceptors work on a per-interface basis (the class name is <*interface_name*>*StubInterceptor* and <*interface_name*>*SkeletonInterceptor*) and are twofold:

- Skeleton interceptors allow to attach coordinators/controllers on provided ports (facets and event sinks),
- Stub interceptors allow to attach coordinators/controllers on required ports (receptacles and event sources).

Finally, some typed wrapping classes are generated. Their goals are twofold:

- First, to code the glue between the client-side interfaces operations and the server-side interfaces operations. For example, for an event named *E1* on an event source named *source1*, the client-side operation is called *push_E1* whereas the server-side operation is called *push_source1* in the case of a monolithic implementation and *push* in the case of a segmented implementation.
- Second, to provide an abstraction to the container of the used implementation strategy. Actually, from the point of view of the container, only the segmented strategy is used. In the case of a monolithic strategy, the executor is wrapped by a “fake” executor locator implementation class named <*component_name*>*MonolithicWrapper*.

The next figure shows the generated Java skeletons classes diagram:

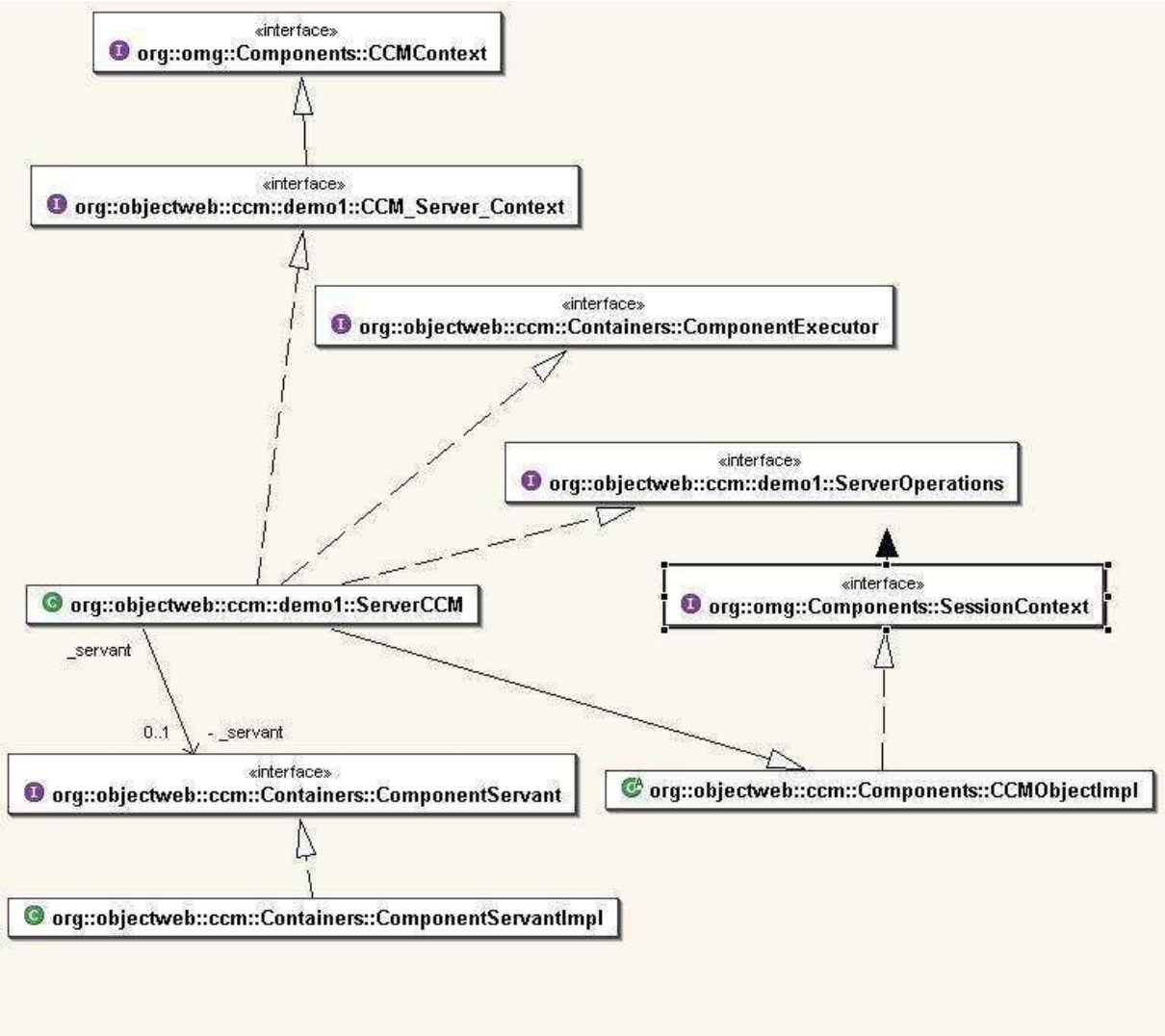

Figure 39 - The Generated Java Extended Skeleton Classes Diagram

3.4.2 The Generated IDL to Java Mappings

As seen in the second paragraph of this document, the OpenCCM platform provides a generator for IDL3 to IDL2 mappings. The IDL compiler provided by the used ORB with OpenCCM, generates Java mappings used by the component client (the client application) and the component implementer (the component executor).

The client application uses the client-side OMG IDL2.x mappings which are implemented by the client stubs.

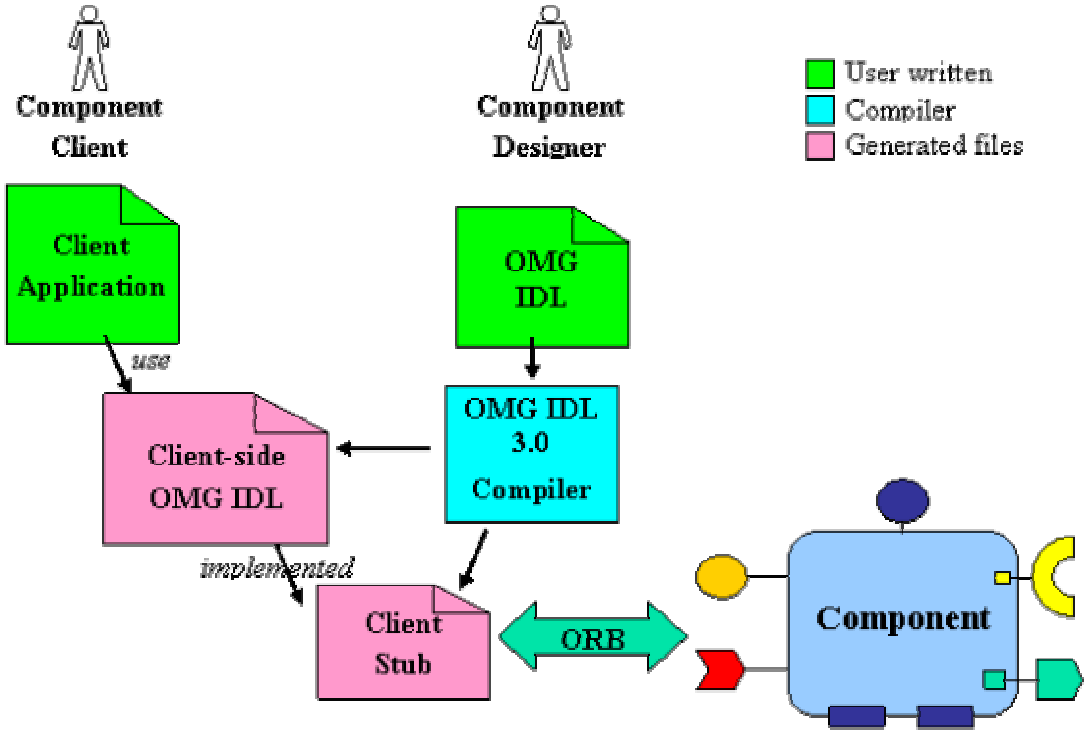

Figure 40 - OMG IDL2.x Mappings from Client View Point

- The *Client* component, that uses the *Display* service provided by the *Server* component, is mapped to a *Client* interface inheriting from *Component::CCMObject*.

This *Component::CCMObject* interface is generated in the *org.omg.Components* package by the ORB's IDL compiler from the *Components.idl* description source file located in the *src\idl* directory in OpenCCM distribution.

- The *to_server* receptacle of *Client* component is mapped to operations for connecting, disconnecting and obtaining the associated reference.
- The *Client* component's home is mapped to three interfaces which are: *ClientHomeExplicit* for explicit operations user-defined inheriting from *Component::CCMHome*, *ClientHomeImplicit* and *ClientHome*.

The *CCMHome* interface is generated in the *org.omg.Components* package from the *Components.idl* file.

OMG IDL3

```
component Client

    {

        attribute string the_name;

        uses Display to_server;

    };
```

OMG IDL2.x Client-Side Mappings

```
interface Client : ::Components::CCMObject

    {

        attribute string the_name;

        void connect_to_server(in ::demo1::Display connexion) raises(...);

        ::demo1::Display disconnect_to_server() raises(...);

        ::demo1::Display get_connection_to_server();

    };
```

OMG IDL3

```
home ClientHome manages Client

    {

    };
```

OMG IDL2.x Client-Side Mappings

```
interface ClientHomeExplicit : ::Components::CCMHome    {    };

interface ClientHomeImplicit : ::Components::KeylessCCMHome

    {

        ::demo1::Client create() raises(...);

    };

interface ClientHome : ::demo1::ClientHomeExplicit,
::demo1::ClientHomeImplicit

    {

    };
```

Figure 41 - Client-Side IDL Mappings for the "Client" Component

Client-side mappings for the *Server* component follow the same rules as for the *Client* component:

OMG IDL3

```
component Server

    {

        attribute string the_name;

        provides Display for_clients;

    };
```

OMG IDL2.x Client-Side Mappings

```
interface Server : ::Components::CCMObject

    {

        attribute string the_name;

        ::demo1::Display provide_for_clients();

    };
```

OMG IDL3

```
home ServerHome manages Server

    {

    };
```

OMG IDL2.x Client-Side Mappings

```
interface ServerHomeExplicit : ::Components::CCMHome {  };

interface ServerHomeImplicit : ::Components::KeylessCCMHome

    {

        ::demo1::Server create() raises(...);

    };

interface ServerHome : ::demo1::ServerHomeExplicit,
::demo1::ServerHomeImplicit

    {

    };    };
```

Figure 42 - Client-Side IDL Mappings for the "Server" Component

The next figures show the class diagram for *Client* or *Server* components and homes, from client view point:

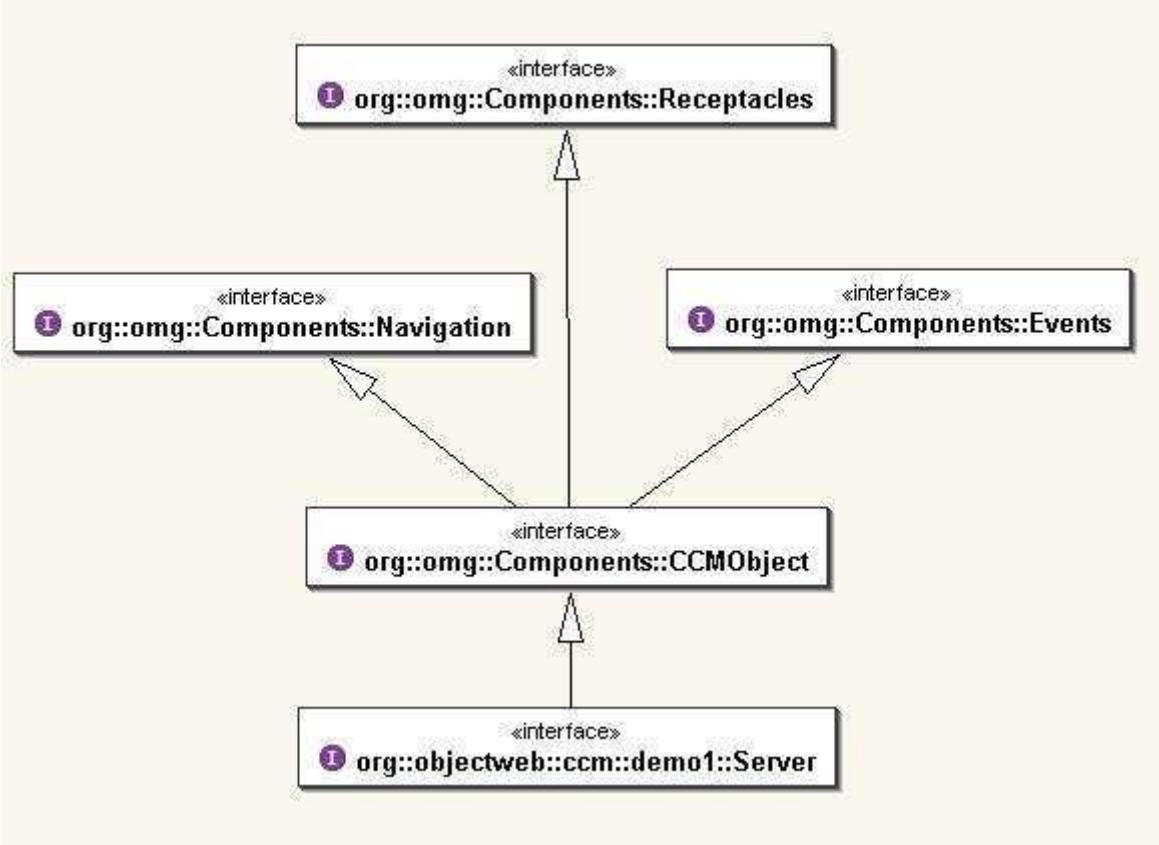

Figure 43 - Component Class Diagram from Client View Point

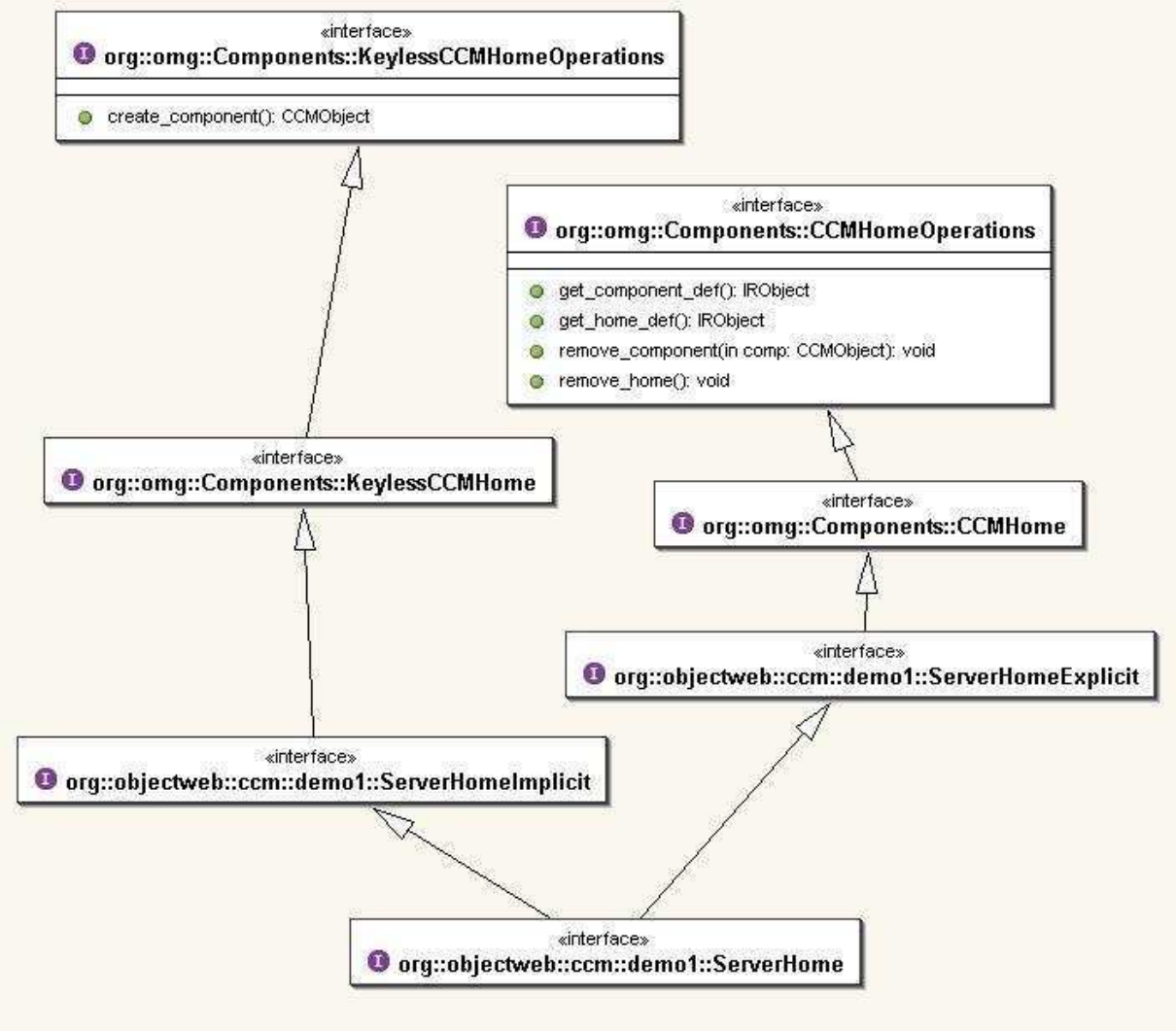

Figure 44 - Home Class Diagram from Client View Point

Server-side OMG IDL mappings provide local interfaces generated by the ORB Java to IDL compiler and used by the component implementer, as illustrated in the following picture:

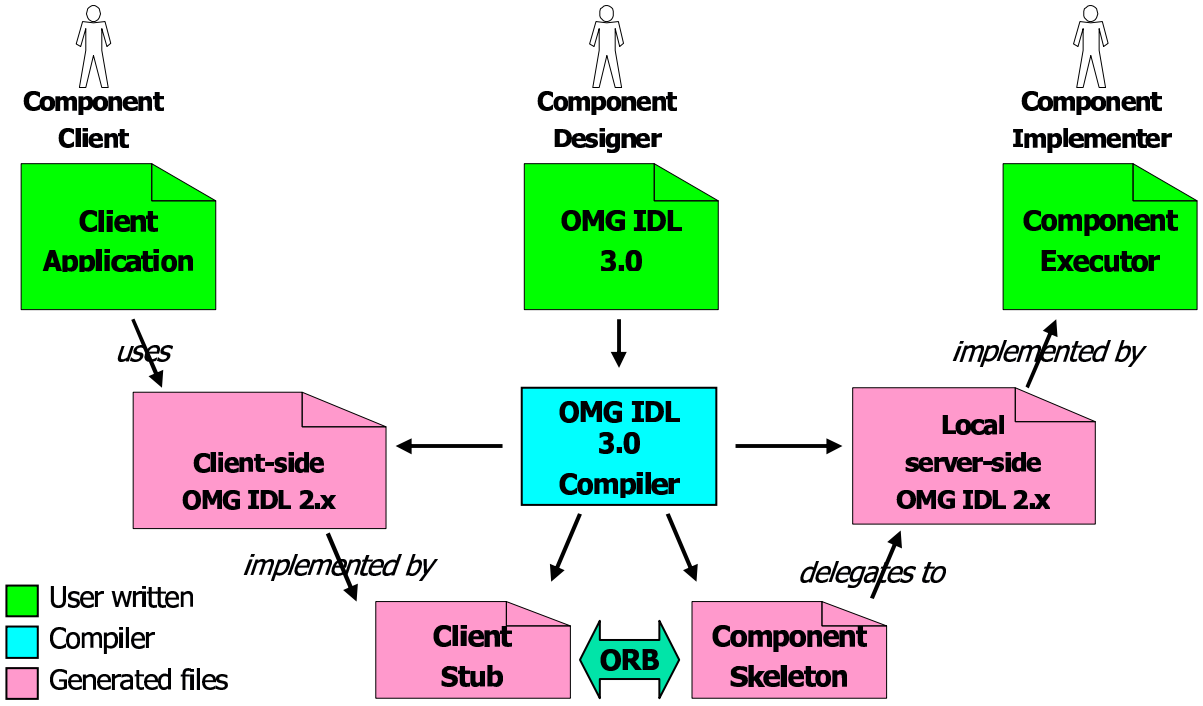

Figure 45 - OMG IDL2.x Mappings from Server View Point

- The *Client* component is mapped to three local interfaces, which are:

- *CCM_Client_Executor* : the main component executor interface inheriting from *Components::EnterpriseComponent* (interface located in *org.omg.Components* package)
- *CCM_Client*: the monolithic component executor interface
- *CCM_Client_Context*: the component specific context interface which provide operation to access the *to_server* component receptacle.

- The *Client* component's home is mapped to three local interfaces, which are:

- *CCM_ClientHomeExplicit* : for explicit operations user-defined inheriting from *Components:: HomeExecutorBase*,
- *CCM_ClientHomeImplicit*: for implicit operations generated, and
- *CCM_Client_Context*: the component specific context interface which provide operation to access the *to_server* component receptacle.

OMG IDL3

```
/**  The Client component type.     */

    component Client

    {

        attribute string the_name;
```

```
        /**
         * The receptable to_server to connect the Client component
         * to a Display object or facet reference.
         */
        uses Display to_server;
    };
```

OMG IDL2.x Server-Side Mappings

```
local interface CCM_Client_Executor : ::Components::EnterpriseComponent
    {
        attribute string the_name;
    };
local interface CCM_Client : ::demo1::CCM_Client_Executor
    {
    };
local interface CCM_Client_Context : ::Components::CCMContext
    {
        ::demo1::Display get_connection_to_server();
    };
```

OMG IDL3

```
/**
     * Simple home for instantiating Client components.
     */
    home ClientHome manages Client
    {
    };
```

OMG IDL2.x Server-Side Mappings

```
local interface CCM_ClientHomeExplicit : ::Components::HomeExecutorBase
    {
    };
local interface CCM_ClientHomeImplicit
```

```
    {

    ::Components::EnterpriseComponent create() raises(...);

    };

local interface CCM_ClientHome : ::demo1::CCM_ClientHomeExplicit,
::demo1::CCM_ClientHomeImplicit

    {

    };
```

Figure 46 - Server-Side IDL Mappings for the "Client" Component

- The *Server* component is mapped to three local interfaces, which are:
- *CCM_Server_Executor* : the main component executor interface inheriting from *Components::EnterpriseComponent* (interface located in *org.omg.Components* package)
- *CCM_Server*: the monolithic component executor interface which provide operation to obtain *for_clients* facet executor
- *CCM_Server_Context*: the component specific context interface.

- The *Server* component's home is mapped to three local interfaces, which are:
- *CCM_ServerHomeExplicit* : for explicit operations user-defined inheriting from *Components:: HomeExecutorBase*
- *CCM_ServerHomeImplicit*: for implicit operations generated
- *CCM_Server_Context*: the component specific context interface which provide operation to access the *to_server* component receptacle.

OMG IDL3

```
component Server

    {

        attribute string the_name;

        provides Display for_clients;

    };
```

OMG IDL2.x Server-Side Mappings

```
local interface CCM_Server_Executor : ::Components::EnterpriseComponent

    {

        attribute string the_name;

    };
```

```
local interface CCM_Server : ::demo1::CCM_Server_Executor

    {

        ::demo1::CCM_Display get_for_clients();

    };

local interface CCM_Server_Context : ::Components::CCMContext

    {

    };
```

OMG IDL3

```
home ServerHome manages Server

    {

    };
```

OMG IDL2.x Server-Side Mappings

```
local interface CCM_ServerHomeExplicit : ::Components::HomeExecutorBase

    {

    };

local interface CCM_ServerHomeImplicit

    {

        ::Components::EnterpriseComponent create() raises(...);

    };

local interface CCM_ServerHome : ::demo1::CCM_ServerHomeExplicit,
::demo1::CCM_ServerHomeImplicit

    {

    };
```

Figure 47 - Server-Side IDL Mappings for the "Server" Component

Server-side CIDL mappings provide executor interfaces generated by the ORB Java to IDL compiler. Here are these CIDL to IDL mappings for our *Client* and *Server* component

CIDL Declarations for the *Client* component

```
composition session ClientSessionComposition

    {

        home executor HomeImpl
```

```
        {
            implements ClientHome;
            manages ComponentImpl;
        };
    };
```

CIDL Mappings for the *Client* component

```
module ClientSessionComposition
    {
        local interface CIF_HomeImpl : ::demo1::CCM_ClientHome
        {
        };
        local interface CIF_SegmentBase
        {
            ::demo1::CCM_Client_Context get_context();
            ::demo1::ClientSessionComposition::CIF_ComponentImpl
get_main_segment();
        };
        local interface CIF_ComponentImpl
                        : ::demo1::ClientSessionComposition::CIF_SegmentBase,
                          ::Components::ExecutorLocator,
                          ::Components::SessionComponent,
                          ::demo1::CCM_Client_Executor
        {
        };
      };
```

CIDL Declarations for the *Server* component

```
composition session ServerSessionComposition
    {
        home executor HomeImpl
        {
```

```
            implements ServerHome;

            manages ComponentImpl;

        };

    };
```

CIDL Mappings for the *Server* component

```
module ServerSessionComposition

    {

        local interface CIF_HomeImpl

                        : ::demo1::CCM_ServerHome

        {

        };

        local interface CIF_SegmentBase

        {

            ::demo1::CCM_Server_Context get_context();

            ::demo1::ServerSessionComposition::CIF_ComponentImpl
get_main_segment();

        };

        local interface CIF_ComponentImpl

                        : ::demo1::ServerSessionComposition::CIF_SegmentBase,

                          ::Components::ExecutorLocator,

                          ::Components::SessionComponent,

                          ::demo1::CCM_Server_Executor,

                          ::demo1::CCM_Display

        {

        };

    };
```

Figure 48 - Client-side and Server-side CIDL Mappings

The following figures show the class diagram for *Client* or *Server* components and homes, from server view point:

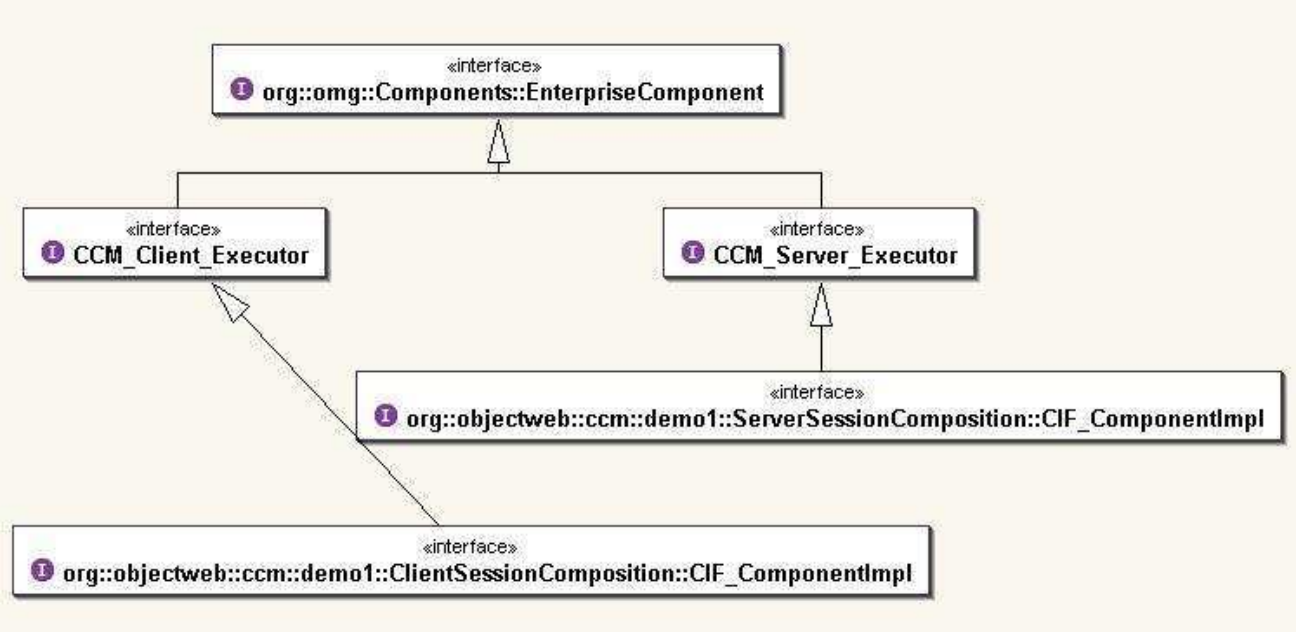

Figure 49 - Component Class Diagram from Server View Point

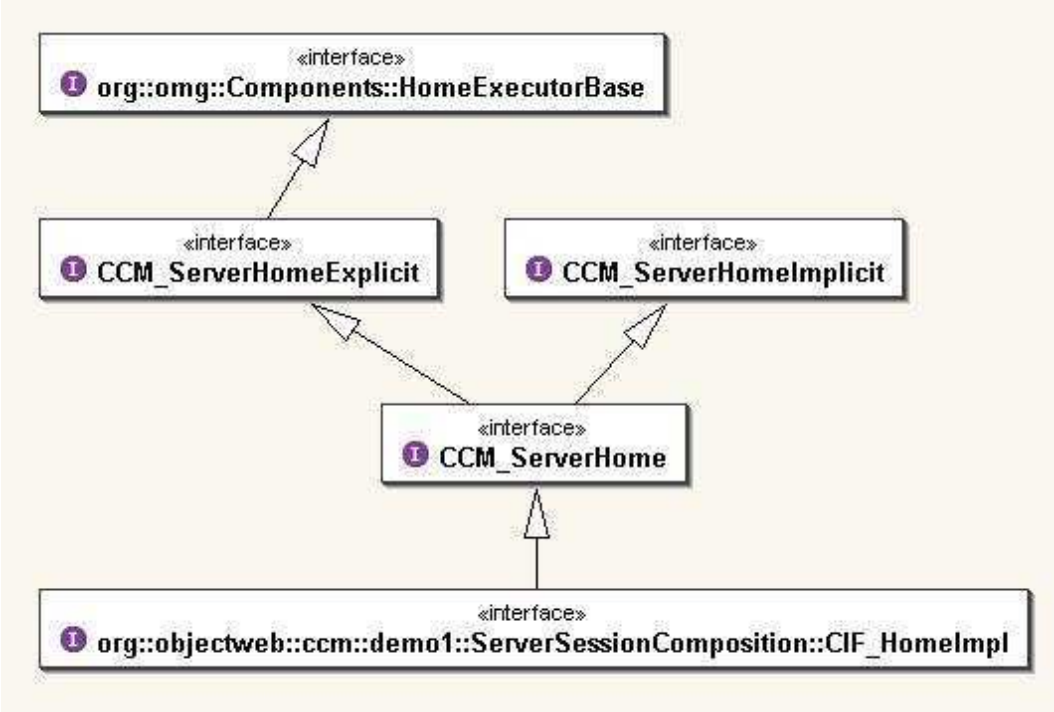

Figure 50 - Home Class Diagram from Server View Point

Finally, the generated client-side and server-side Java mappings for our simple CCM client / server application, are (for the *server* component for instance, replace *server* per *client* to obtain mappings for the *Client* component):

```
\org\objectweb\ccm\demo1
                |   CCM_Display.java
                |   CCM_DisplayHelper.java
                |   CCM_DisplayHolder.java
                |   CCM_DisplayOperations.java
                |   CCM_Server.java
                |   CCM_ServerHelper.java
                |   CCM_ServerHolder.java
                |   CCM_ServerHome.java
                |   CCM_ServerHomeExplicit.java
                |   CCM_ServerHomeExplicitHelper.java
                |   CCM_ServerHomeExplicitHolder.java
                |   CCM_ServerHomeExplicitOperations.java
                |   CCM_ServerHomeHelper.java
                |   CCM_ServerHomeHolder.java
                |   CCM_ServerHomeImplicit.java
                |   CCM_ServerHomeImplicitHelper.java
                |   CCM_ServerHomeImplicitHolder.java
                |   CCM_ServerHomeImplicitOperations.java
                |   CCM_ServerHomeOperations.java
                |   CCM_ServerOperations.java
                |   CCM_Server_Context.java
                |   CCM_Server_ContextHelper.java
                |   CCM_Server_ContextHolder.java
                |   CCM_Server_ContextOperations.java
                |   CCM_Server_Executor.java
                |   CCM_Server_ExecutorHelper.java
                |   CCM_Server_ExecutorHolder.java
                |   CCM_Server_ExecutorOperations.java
                |   Display.java
                |   DisplayHelper.java
                |   DisplayHolder.java
                |   DisplayOperations.java
                |   DisplayPOA.java
                |   DisplayPOATie.java
                |   Server.java
                |   ServerHelper.java
                |   ServerHolder.java
                |   ServerHome.java
                |   ServerHomeExplicit.java
                |   ServerHomeExplicitHelper.java
                |   ServerHomeExplicitHolder.java
                |   ServerHomeExplicitOperations.java
                |   ServerHomeExplicitPOA.java
```

```
|   ServerHomeExplicitPOATie.java
|   ServerHomeHelper.java
|   ServerHomeHolder.java
|   ServerHomeImplicit.java
|   ServerHomeImplicitHelper.java
|   ServerHomeImplicitHolder.java
|   ServerHomeImplicitOperations.java
|   ServerHomeImplicitPOA.java
|   ServerHomeImplicitPOATie.java
|   ServerHomeOperations.java
|   ServerHomePOA.java
|   ServerHomePOATie.java
|   ServerOperations.java
|   ServerPOA.java
|   ServerPOATie.java
|   _DisplayStub.java
|   _ServerHomeExplicitStub.java
|   _ServerHomeImplicitStub.java
|   _ServerHomeStub.java
|   _ServerStub.java
|
\---ServerSessionComposition
        CIF_ComponentImpl.java
        CIF_ComponentImplHelper.java
        CIF_ComponentImplHolder.java
        CIF_ComponentImplOperations.java
        CIF_HomeImpl.java
        CIF_HomeImplHelper.java
        CIF_HomeImplHolder.java
        CIF_HomeImplOperations.java
        CIF_SegmentBase.java
        CIF_SegmentBaseHelper.java
        CIF_SegmentBaseHolder.java
        CIF_SegmentBaseOperations.java
```

Figure 51 - Generated Java Mappings for Our Simple CCM Application

3.4.3 Implementation Classes and Dependencies with OpenCCM Runtime

The *Client* component uses the *Display* interface provided by the *Server* component. Considering a monolithic implementation (all component features are provided by the same class):

- The user-written *ClientImpl* class extends the *org.omg.CORBA.LocalObject* class and implements the generated *CCM_Client* local interface.

- The user-written *ServerImpl* class extends the *org.omg.CORBA.LocalObject* class and implements the generated *CCM_Server* local interface and the *CCM_Display* local interface.

Considering a CIF implementation:

- The user-written *ClientImpl* implementation class extends the *ClientSessionComposition::ComponentImpl* class (the *Client* component executor implementation skeletons) which implements the generated *ServerSessionComposition::CIF_ ComponentImpl* local interface.
- The user-written *ServerImpl* implementation class extends the *ServerSessionComposition::ComponentImpl* class (the *Server* component executor implementation skeletons) which implements the generated *ServerSessionComposition::CIF_ ComponentImpl* local interface. *ServerImpl* have to implement the local *CCM_Display* interface.

The next figure shows the diagram classes for our *Client* and *Server* user implementation classes:

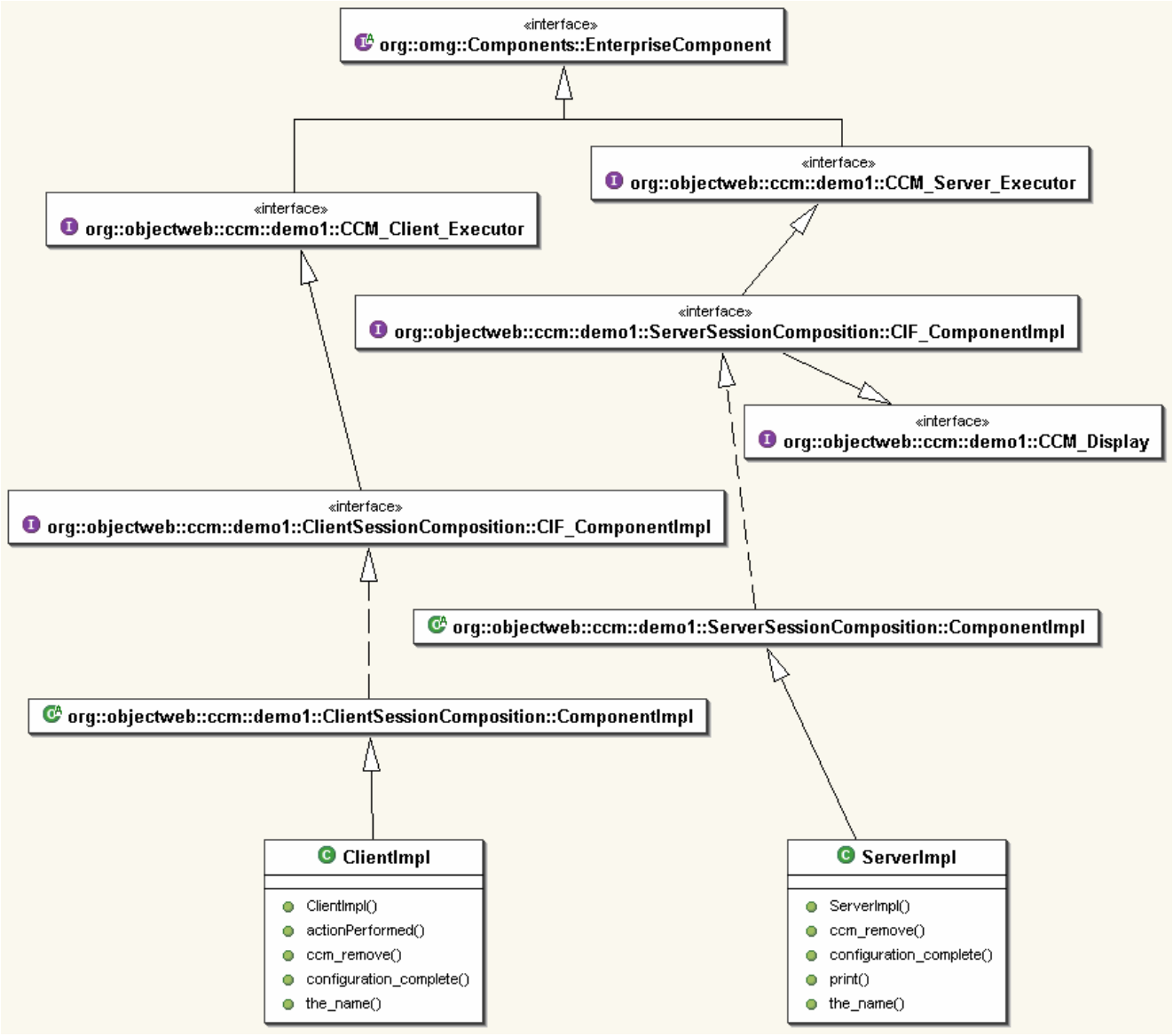

Figure 52 - Class Diagram for Generated and Component Implementation Classes

3.5 The Inside OpenCCM Big Picture

The following figure finally describes what the OpenCCM platform produce from IDL and CIDL description files:

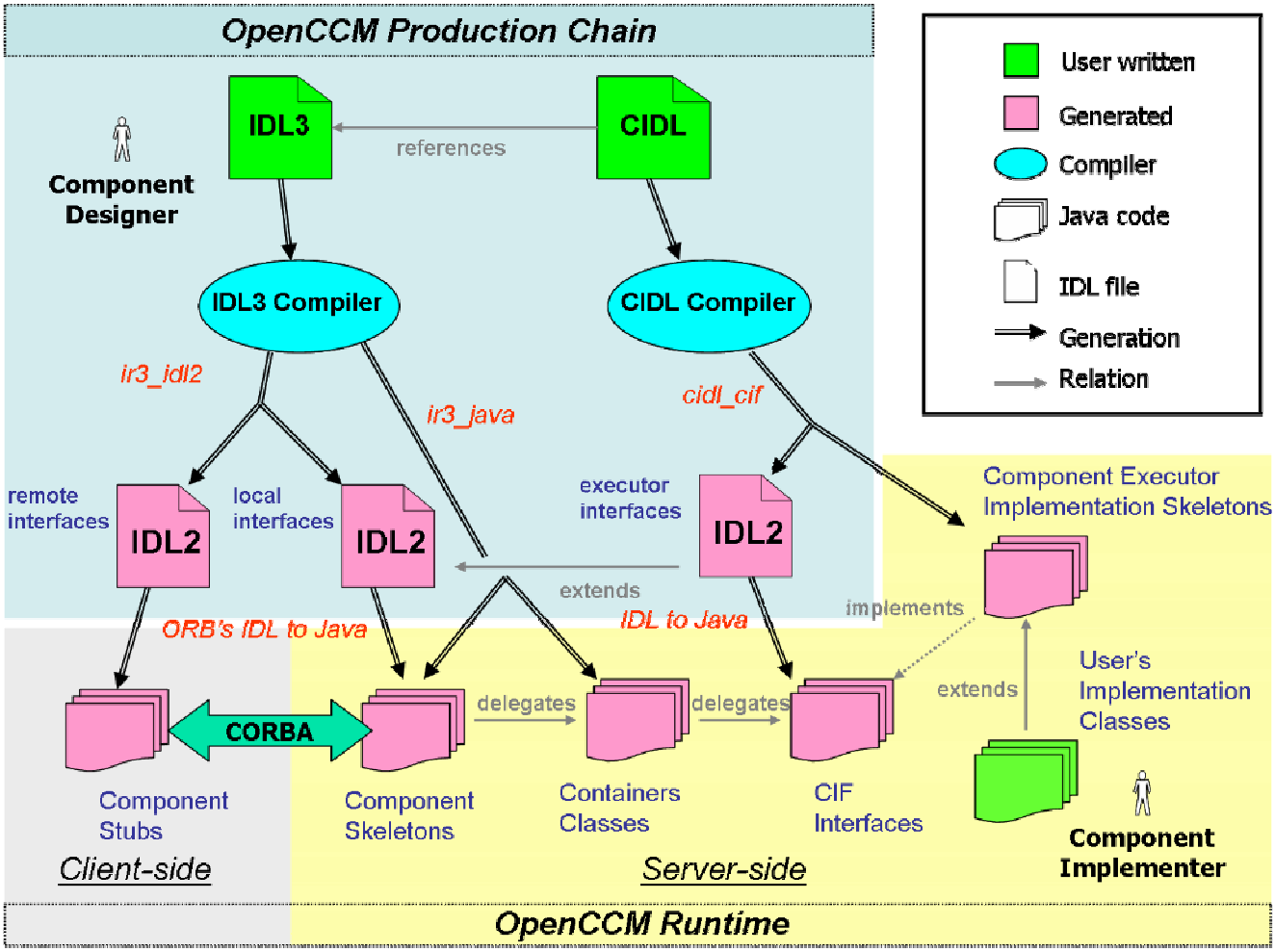

Figure 53 - Inside OpenCCM Big Picture

The OpenCCM production chain allows, through the IDL3 and CIDL compilers and command scripts to generate:

- IDL2 remote interfaces for the client-side (*ir3_idl2*),
- IDL2 local interfaces for the server-side (*ir3_idl2*),
- and IDL2 executor interfaces (*cidl_cif*)

Also, the production chain generates classes used by the OpenCCM runtime:

- component skeletons (*ir3_java*)
- container classes (*ir3_java*)
- and component executor implementation skeletons (*cidl_cif*)

All necessary CORBA 2 stubs and skeletons and CIF interfaces are generated using the user's favorite ORB's IDL to Java compiler.

The user or developper only have to concentrate on its business code to write implementation classes. Note that the figure 44 also shows dependancies (inheritance, delegation and implementation) between OpenCCM runtime classes, which illustrate the process followed by a client request on a component.

4 OPENCCM DEPLOYMENT AND EXECUTION INFRASTRUCTURE

The OpenCCM 0.6 execution chain consists of various scripts. Some of them set up the environment and run background processes like the name service, Java Component Servers, and so on; others really initiate the deployment and execution of a CCM application.

4.1 Installation of the OpenCCM Deployment Infrastructure

The *ccm_install* command installs OpenCCM the deployment infrastructure by creating a *OpenCCM_CONFIG_DIR* under the generated <ORB name> directory with a Windows-like OS, or under the user home dir with a unix-like OS. It also creates a *ComponenServers* subdirectory that will contain the archives of software packages installed by the deployment process.

```
OpenCCM_CONFIG_DIR
|   |   NameService.IOR
|   |   NameService.IOR.tmp
|   |   NameService.PID
|   |
|   \---ComponentServers
```

Figure 54 - The Generated OpenCCM Configuration Directories

4.2 Starting the Name Service of the used ORB

When the Name Service of the used ORB is started, with the *ns_start* command, two files are stored in the *OpenCCM_CONFIG_DIR\<used_ORB_name>* directory. First one, *NameService.IOR*, contains the IOR, and second one, *NameService.PID*, contains the PID of the started process.

This PID is used by the *ns_stop* command to destroy the process and delete associated files.

4.3 The Java Component Server

The Java Component Server is the entity in charge of downloading archive files, and hosting the container. With the *jcs_start* command , one can start a Java Component Server (typically identified as *ComponentServer1* or *ComponentServer2 in the demonstrations*) by running *jcs_start <component_server_name>*. This script command runs the *ServerMain* class which is located in the *org.objectweb.ccm.Deployment* package.

The *ServerMain* class takes a parameter that will be the JCS id, and does the following:

- Creates the server servant by instanciating a new *ServerImpl* object and activates it,
- Obtains the Name Service and binds the server object into it with the provided id,
- Prints the IOR of the component server on standard output. This is forwarded by the script in a <component_server_name>.IOR.

```
+---OpenCCM_CONFIG_DIR
|   |   NameService.IOR
|   |   NameService.IOR.tmp
|   |   NameService.PID
|   |
|   \---ComponentServers
|       |   ComponentServer1.IOR
|       |   ComponentServer1.output
|       |   ComponentServer1.output.tmp
|       |   ComponentServer1.PID
|       |
|       \---ComponentServer1.archive_cache
```

Figure 55 - OpenCCM Generated Component Server Configuration Files

The *org.objectweb.ccm.Deployment* package contains the next files:

```
\---org
|   |  \---objectweb
|   |           +---ccm
|   |           |   +---Deployment
|   |           |   |   |   ComponentInstallationImpl.java
|   |           |   |   |   ComponentServerBase.java
|   |           |   |   |   ComponentServerImpl.java
|   |           |   |   |   ComponentServerLocalImpl.java
|   |           |   |   |   ContainerBase.java
|   |           |   |   |   ContainerImpl.java
|   |           |   |   |   ContainerLocalImpl.java
|   |           |   |   |   ServerImpl.java
|   |           |   |   |   ServerMain.java
|   |           |   |   |   SystemHomeManagerImpl.java
|   |           |   |   |   TheComponentInstallation.java
|   |           |   |   |   TheURLClassLoader.java
```

Figure 56 - Deployment Implementation Package

4.4 What should be done in a deployment process: A simple example

In this section, we will see what should be done in which order during a manual deployment process and the execution of a CCM application. There is also an automated way to deploy components and run the application with the OpenCCM platform. The first way is based on the use of a Java script as the "bootstrap" of the application. The second way, is to use the OpenCCM deploy tool which only uses XML descriptors. We will see this in details in the next paragraph, "*4.5 The OpenCCM Deploy Tool*".

Once one have designed his CCM application, generated stubs and skeletons, written implementation files containing business code of the application, compiled it and built an archive, time is now to deploy components and run the application. This means to attach virtual component locations to physical nodes and install components and assemblies to particular nodes on the network by starting the deployment process.

4.4.1 Initializing the ORB and the OpenCCM Runtime

In a deployment script, the first thing to do is the initialization of the OpenCCM runtime by calling the *init* method of the *org.objectweb.ccm.Components.Runtime* class:

```
// Init the OpenCCM Components Runtime.

org.omg.CORBA.ORB orb = org.objectweb.ccm.Components.Runtime.init(args);
```

4.4.2 Obtaining the Name Service

To start the Name Service, we call the *resolve_initial_reference* method of the instantiated *orb* CORBA object:

```
org.omg.CORBA.Object obj =

            orb.resolve_initial_references("NameService");

org.omg.CosNaming.NamingContext nc =

            org.omg.CosNaming.NamingContextHelper.narrow(obj);
```

4.4.3 Obtaining Component Servers

Assuming component servers *ComponentServer1* and *ComponentServer2* are started and registered in the started Name Service, we obtain component servers with:

```
// Obtain the component servers.

System.out.println("3.Obtaining Component Servers...");

org.omg.CosNaming.NameComponent[] ncomp =

                              new org.omg.CosNaming.NameComponent[1];

ncomp[0] = new org.omg.CosNaming.NameComponent("ComponentServer1", "");

obj = nc.resolve(ncomp);

org.objectweb.ccm.Deployment.Server server1 =
      org.objectweb.ccm.Deployment.ServerHelper.narrow(obj);

ncomp[0].id = "ComponentServer2";

obj = nc.resolve(ncomp);

org.objectweb.ccm.Deployment.Server server2 =
                  org.objectweb.ccm.Deployment.ServerHelper.narrow(obj);
```

4.4.4 Obtaining Container Homes and Archives

Using *provide_component_servers* and *provide_install* methods of the *org.objectweb.ccm.Deployment.Server* class, we obtain the container homes and archives installers :

```
// Obtain the container homes and archive installators.

      org.omg.Components.Deployment.ComponentServer server1_cs =

server1.provide_component_server();

org.omg.Components.Deployment.ComponentInstallation server1_inst =

server1.provide_install();

      org.omg.Components.Deployment.ComponentServer server2_cs =

 server2.provide_component_server();

      org.omg.Components.Deployment.ComponentInstallation server2_inst =

server2.provide_install();
```

4.4.5 Installing Archives

Now, we have to install the archives of components of our CCM application on the component servers:

```
// Install archives.

server1_inst.install("demo", "./archives/demo.jar");

server2_inst.install("demo", "./archives/demo.jar");

server1_inst.install("openccm_plugins", "./OpenCCM_Plugins.jar");

server2_inst.install("openccm_plugins", "./OpenCCM_Plugins.jar");
```

Archives of the application *demo.jar* and *openccm_plugins.jar* are installed via the network in the *ComponentServers\ComponentServer<1or2>.archive_cache* directory. Archives are locally copied or can be downloaded through HTTP protocol through the network.

4.4.6 Installing a Container on each Server

Then, we instantiate a container on each server, using the *create_container* method of component server object:

```
org.omg.Components.Deployment.Container server1_cont =

                        server1_cs.create_container(config);

org.omg.Components.Deployment.Container server2_cont =

                        server2_cs.create_container(config);
```

4.4.7 Installing Homes

To install homes, we declare the container configuration and use the *install_home* method of instantiated container object by providing the archive name of component type and the entry point:

```
// set the container config as the home config

// i.e. an empty configuration.

container.set_home_configuration(container.get_container_configuration()) ;



// create a port specific config

config=container.find_system_home("PortConfig").create_component(JAVA.null)
;



// set the component ::CORBA::Repository of the component.

config.setComponentUId("IDL:ccm.objectweb.org/demo1/Server:1.0") ;
```

```
// set a list coordinator for the "for_clients" facet of the "Server"

// component

config.setCallCoordinator("for_clients", "ListCoord") ;


// set a trace controller for the "for_clients" facet of the "Server"

// component

config.setCallControllers("for_clients", [ "Trace" ]) ;


// create and set the trace configuration for this facet

trace_config = PropertySet("trace") ;

trace_config.add(StringProperty("filename", "./trace_Server.txt")) ;

config.setCallControllersConfig("for_clients", [ trace_config ]) ;


// set the container config as the parent config.

config.set_parent_configuration(container.get_container_configuration()) ;


// set this config as the component config.

container.set_component_configuration(config) ;

// install the Server home with this container configuration.

value.insert_string(home_config);

config[0].value = value;

org.omg.Components.CCMHome h = server1_cont.install_home("demo1",
"org.objectweb.ccm.demo1.ServerHomeImpl.create_home", new
org.omg.Components.ConfigValue[0]);

ServerHome sh = ServerHomeHelper.narrow(h);
```

4.4.8 Creating and Configuring Components

Simply using the *create* method of home object, we instantiate components and configure them:

```
// Create components.

Server s = sh.create();

Client c1 = ch.create();

Client c2 = ch.create();

Client c3 = ch.create();

// Configure components.

s.the_name("The Server");

c1.the_name("Mathieu");

c2.the_name("Raphael");

c3.the_name("Philippe");
```

4.4.9 Connecting Components

Considering a *Server* component providing a *Display* facet used by a *Client* component through its receptacle, and using the IDL2 to Java client-side mapped operations:

```
// Connect clients and server.

Display for_clients = s.provide_for_clients();

c1.connect_to_server(for_clients);

c2.connect_to_server(for_clients);

c3.connect_to_server(for_clients);
```

4.4.10 Configuration Completion

The *configuration_complete( )* method of component implementations completes the deployment process, for each instantiated components:

```
// Configuration completion.

s.configuration_complete();

c1.configuration_complete();

c2.configuration_complete();

c3.configuration_complete();
```

The deployment process is now finished and our CCM application is ready to be used. In fact, switching components to the *configuration_complete* state trully starts the execution of the application.

4.5 The OpenCCM Deploy Tool

The CCM specification allows the use of XML based files to describe assemblies of components instances (this is the assembly concept) and to describe components archives contents (this is the packaging concept). In OpenCCM 0.6, the reading, analyse, and use of such meta information is done via an automated deployment tool, which avoids to manually instantiate homes and components, operate with the Naming Service, and connect instances together.

4.5.1 Inside the Deployment Tool Process

The *ccm_deploy* script calls the *org.objectweb.ccm.deploytool* package, forwarding the provided parameter that points to the component assembly descriptor (CAD) file of the application.

DeploymentApplication uses a *CommandLine* framework, which allows exceptions interception, console display of messages, and so on. The deployment process itself is located in *DeploymentProcess*, which handles the unmarshalled meta information, and delegates the setting up of components partition and the connection of these instances.

The figure 56 shows how partitionning and connecting are separated, in order that the deployment does not follow a too much linear logic (and avoid time lost). The *Connector* gets initialized and ready to make connections by pre-reading the meta information and feeding an hashtable. The *Partition* analyses the placements definitions. Once it deploys a component instance, it notifies the *Connector*, so that involved connections are activated (one party ready) or done (two parties ready).

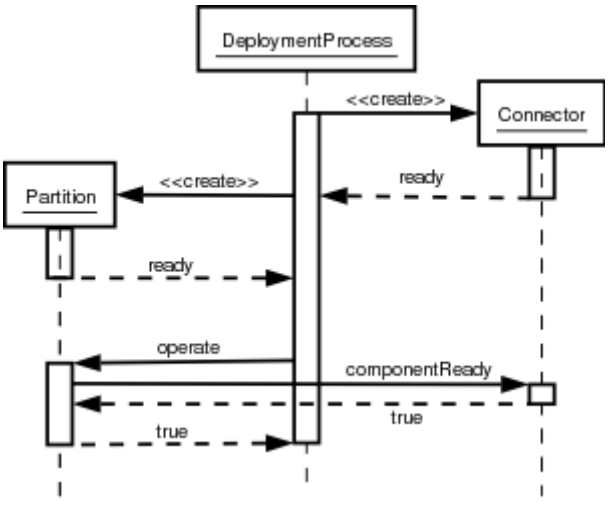

Figure 57 - Sequence Diagram of the Automated Deployment Process

4.5.2 Writing XML Meta Information

Current version of OpenCCM's deployment tool does not fully support the CCM's XML formalism. Focus has been put on the essentials elements, which allow rudimentary deployments in small infrastructure contexts. Let's see which restrictions apply when using meta information for deployment purposes in OpenCCM 0.6.

4.5.2.1 Pointing Other Files

In the description of assemblies and components, some elements are used to point other files such as descriptors, code archives, and so on. For example, a component assembly descriptor (CAD) points to the component software descriptors (CSD) in the <*componentfiles*> part. It points to property file descriptors (CPF) too, in order to apply specific properties to instantiated components. In a same manner, the CSD points to code archives implementing the component.

In CCM, these references can be URI, local files names, or direct environment resource. In OpenCCM 0.6, they can only be done using a <*fileinarchive*> element. Using <*link*> or <*codebase*> is not supported, that means all the meta information must be found locally in the assembly's META-INF/ folder.

4.5.2.2 Structuring the Distributed Application

The assembly's partitioning can contain collocations, so that components can be located on same host or same process. OpenCCM 0.6 does support such definitions, with the <*hostcollocation*> and <*processcollocation*> elements. The placements definitions for components homes can be provided inside these groups, or individually in case the home is alone on a destination.

The <*destination*> element, which format is determined by the deployment tool itself, must contain a NS registered name for a component server. That means each involved Java Component Server in OpenCCM 0.6 must be registered to the naming service (this is something done in case you use the provided *jcs_start* script).

The <*executableplacement*> element is not supported. Each component should be created through the usage of a home, as the deployment tool of OpenCCM 0.6 can not deploy executables.

When using a component, several implementations can be provided within a descriptor (CSD), but in the context of rudimentary applications, each component software has only one implementation. Pointing out components descriptors (CSD) in an assembly (CAD) should allow to select which implementation of the component may be used in the described application. In the current version of our deployment tool, such selection is not possible. OpenCCM 0.6 will always use the first implementation described in a CSD file.

4.5.2.3 Connecting Components and Applying properties

OpenCCM 0.6 fully supports components connections decsription within an assembly file, but does not support <*proxyhome*> elements (and consequently <*connecthomes*>).

An application should use interfaces or events, but it must always locally provide involved ports (and components): In other words, only basic use of <*usesport*>, <*providesport*>, <*consumesport*>, and <*publishesport*> elements is possible, with <*...identifier*> elements. Usage of the Naming Service through <*findby*> elements is not supported.

When applying properties to components, such as identifying names, the deployment tool only looks in the component instance definition, in the assembly. It should be possible, according to the CCM specification, to have several kinds of properties: at design level, for any implementation, at implementation level (depending on the language), and at instantiation level. In OpenCCM 0.6, it is only possible to define a properties file descriptor (CPF) within a <*componentinstantiation*> element of the assembly.

4.6 Deploying and Executing the simple CCM Client / Server Example

In this section, we see with the previous simple CCM Client / Server example, how to write XML descriptors to deploy the application and how they are used by the deploy tool.

The XML Component Assembly Descriptor (*demo1.cad*) uses the component assembly DTD located in the *src\dtd\ccm* directory of OpenCCM distribution and does the following:

- References component software descriptors for each component type *Client* and *Server*, like this:

```
<componentfiles>
    <componentfile id="Client">
      <fileinarchive name="client.csd">
      </fileinarchive>
    </componentfile>
    <componentfile id="Server">
      <fileinarchive name="server.csd">
      </fileinarchive>
    </componentfile>
 </componentfiles>
```

Figure 58 - Referencing Component Sofware Descriptors in CAD

- Defines the partitioning using <*homeplacement*>. The registered Component Server in the Name Service is mentionned with the <*destination*> tag. The <*componentimplref* > tag allows to select the associated package

```
<homeplacement cardinality="1" id="ServerHome">
      <componentfileref idref="Server"/>
      <componentimplref idref="ServerImpl"/>
      <registerwithhomefinder name="OpenCCM/ServerHome"/>
      <registerwithnaming name="OpenCCM/ServerHome"/>
      <componentinstantiation id="Server">
        <componentproperties>
          <fileinarchive name="server.cpf">
          </fileinarchive>
        </componentproperties>
      </componentinstantiation>
      <destination>ComponentServer1</destination>
</homeplacement>
```

Figure 59 - Defining Partitioning for Distributed Deployment

- Defines connections between the adequate *Client* and *Server* component ports: the *Client* component *to_server* receptacle connects to the the *for_clients* facet *Server* component.

```
<connectinterface>
      <usesport>
        <usesidentifier>to_server</usesidentifier>
        <componentinstantiationref idref="Client 1"/>
      </usesport>
      <providesport>
        <providesidentifier>for_clients</providesidentifier>
        <componentinstantiationref idref="Server"/>
      </providesport>
</connectinterface>
```

Figure 60 – Defining the Connections

To start the deployment process, using user-written XML descriptors as arguments, we use the *ccm_deploy* OpenCCM command script. This run the deploy tool, which is in the *org.objectweb.ccm.deploytool* package.

5 COMPILATION AND INSTALLATION OF OPENCCM PLATFORM

5.1 Compilation of OpenCCM Platform

The *build* command which compile OpenCCM sources does the following (see the *build.xml* file for more details):

i. Create directories (see above)
ii. Generate stubs for the OMG Interface Repository 3.0
iii. Generate CORBA stubs for OMG IDL files
iv. Compile stubs for the OMG Interface Repository 3.0
v. Build the initial OpenCCM JAR archive
vi. Compile local interfaces
vii. Generate the OMG IDL3/PSDL/CIDL parser files
viii. Compile Java packages for the OpenCCM platform
ix. Build the final OpenCCM JAR archives
x. Create binary scripts for Unix or Windows OS
xi. And prepare OpenCCM demonstrations

The result of the compilation is the following subdirectories in the *<name of the used ORB>* directory:

- *bin:* OpenCCM command scripts
- *classes*: all compiled OpenCCM Java packages (for runtime and production chain)
- *doc*: documentation files generated using doxygen tool
- *dtd*: DTD of the deploy tool
- *generated*: generated Java files from OpenCCM IDL description source files
- *lib*: this subdirectory contains Java archives (jar) of OpenCCM distribution and other packages and tools
- *template*: needed template file used by OpenCCM generators and template for generated Java skeletons
- *xml*: this subdirectory contains XML configuration files for OpenCCM commands (used by the Java launcher) and XML files of the deploy tool

5.2 Installation of OpenCCM Platform

To install the OpenCCM platform, simply type “build install”. This will create a *build* directory that contains the following subdirectories:

```
\OpenCCM-0.6\build
+---bin
|       ccm_deinstall.bat
|       ccm_deploy.bat
|       ccm_install.bat
|       ....
```

→ OpenCCM binary scripts of the user' OS

```
|
+---idl
|       Components.idl
|       CosTransactions.idl
|       Deployment.idl
|       IFR_3_0.idl
|       ...
```

→ OpenCCM binary IDL source files

```
+---lib
|       apollon.jar
|       dtdparser114.jar
|       jdbc2_0-stdext.jar
|       jdom.jar
|       jidlscript.jar
|       jta_1.0.1.jar
|       Launcher.jar
|       LICENSE.txt
|       NamingContext.jar
|       OpenCCM.jar
|       OpenCCM_Plugins.jar
|       openorb_ots-1.3.0.jar
|       velocity-1.3.1-rc2.jar
|       velocity-dep-1.3.1-rc2.jar
|       xerces.jar
|       zeus.jar
```

→ All OpenCCM Java archives, packages and tools

```
\---templates
        cidl.vm
        cif.vm
        common.vm
        idl3.template
        idl3.vm
        java.vm
        java_impl_common.template
        ...
```

→ OpenCCM template file for generators

Figure 61 - The build Directory of Installed OpenCCM Platform

6 ANNEXES

6.1 OpenCCM Generators Class Diagram

6.1.1 The Basic Generator Class Diagram

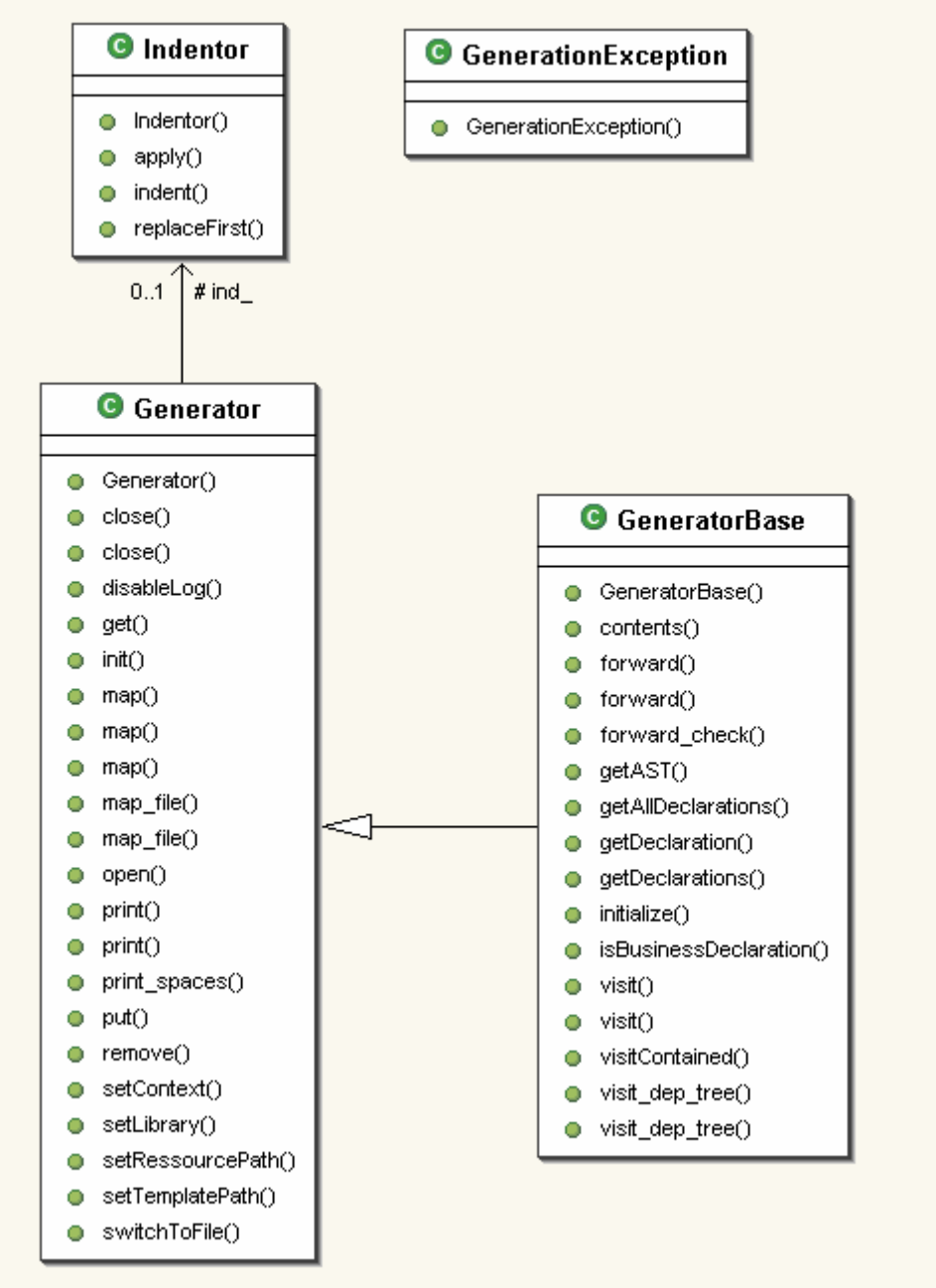

Figure 62 - Basic Generator Class Diagram

6.1.2 The IDL Generator Class Diagram

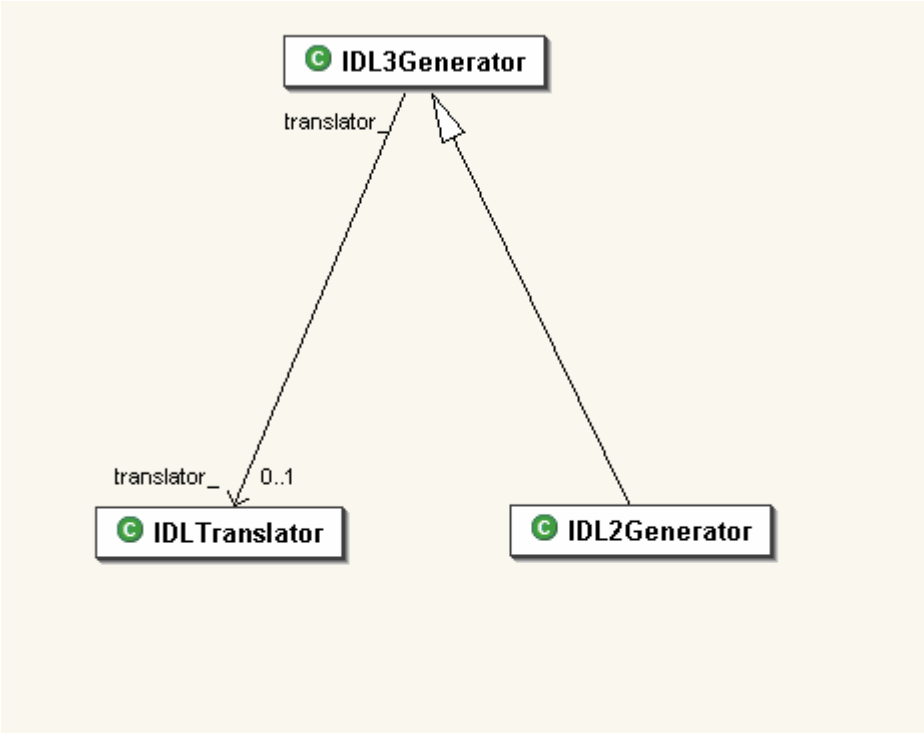

Figure 63 - IDL Generator Class Diagram

6.1.3 The PSDL Generator Class Diagram

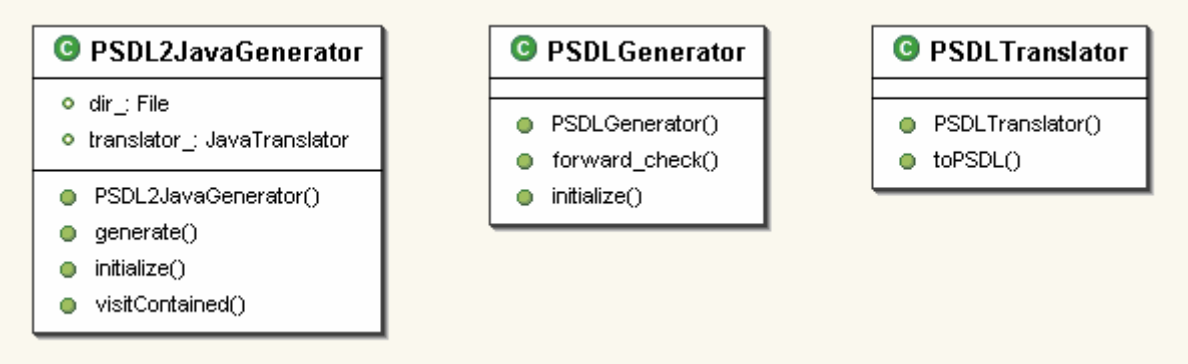

Figure 64 - PSDL Generator Class Diagram

6.1.4 The CIDL Generator Class Diagram

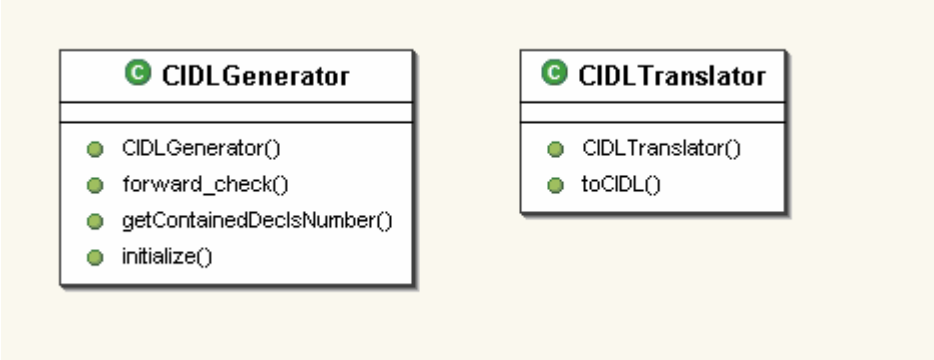

Figure 65 - CIDL Generator Class Diagram

6.1.5 The CIF Generator Class Diagram

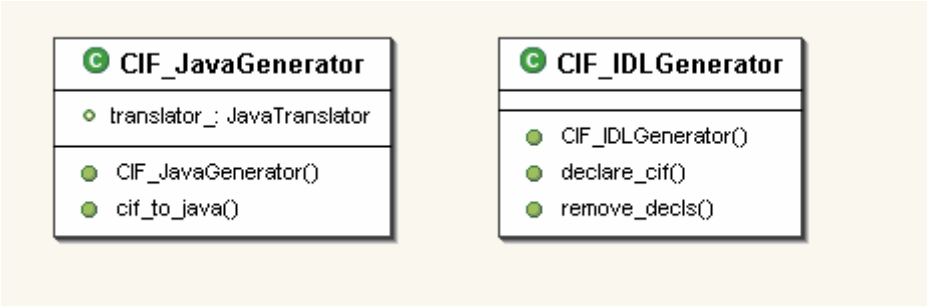

Figure 66 - CIF Generator Class Diagram

6.2 Trace Service in OpenCCM (using Monolog Framework, Objectweb)

6.2.1 Monolog Framework concepts

Monolog is a framework develop in the ObjectWeb's code base (http://www.objectweb.org/monolog/index.html). It defines an abstraction to use logging in applications. Its two purposes are :

- wraps the logging system really used (for example log4j)
- separate Configuration and Instrumentation preoccupation

Monolog defines a system based upon some concepts :

- Logger (TopicalLogger) :

  Objects to use for tracing.

  There are associated to, at least, one Topic.

  Topics associated to a logger are keys to get this logger.

  Loggers are hierarchically connect (as Java namespace), so all messages send to a logger are routed to all its loggers ancestor.

- Topics :

  Strings associated to Loggers in order to manipulate them.

  Defined a Trace namespace.

- Handler :

  Represents outputs of Loggers.

  One logger could have a lot of handler associated to it, so a lot of different output.

- Level :

  Defines different level of importance in Traces.

  These Levels are ordered.

  Each Logger has its own Level of tracing.

  Each message of trace must be categorize by a Level.

  So, Loggers only trace message of its own level and all greater Level.

  By default Monolog defines some BasicLevel :

  - INHERIT < DEBUG < INFO < WARM < ERROR < FATAL.

  The following table describes the meaning of the five predefined levels.

Level name	Details
FATAL	In general, FATAL messages should describe events that are of considerable importance and which will prevent continuation of the program execution. They should be intelligible to end users and to system administrators.
ERROR	The ERROR level designates error events that might still allow the application to continue running.

WARN	In general, WARN messages should describe events that will be of interest to end users or system managers, or which indicate potential problems.
INFO	The INFO level designates informational messages that highlight the progress of the application at a coarse-grained level.
DEBUG	DEBUG messages might include things like minor (recoverable) failures. Logging calls for entering, returning, or throwing an exception can be traced at this level.
INHERIT	INHERIT is a special Level which permits to specify that a logger must inherit its level from its ancestor.

- Factories :

  LevelFactory, HandlerFactory, LoggerFactory.

  Each factories provide instances for each part of the Monolog system.

See more at: http://www.objectweb.org/monolog/doc/index.html

6.2.2 Use of trace in OpenCCM code base

In each package you have to define a class containing static definitions of loggers you want to use here. By convention you must named this class ***MonologTopics***.

To create that class you need to import:

- ***org.objectweb.util.monolog.api.Logger***
  (the Logger API from Monolog)
- ***org.objectweb.util.monolog.provider.LoggerProviderSingleton***
  (the platform Topic provider static singleton class)

The *LoggerProviderSingleton* wrapped a *LoggerProvider* to ensure that only one instance is used in a JVM. A *LoggerProvider* is used to create *Logger* by Topics which are store in a referential.

These Topics are made using the *org.objectweb.util.monolog.provider.topics.BundleTopics* which get its bundle of *Strings* from the property file *topicsBase.properties* located in the package *org.objectweb.ccm.trace*.

Then, defined a final static boolean. It is used to enable trace specific code insertion at compilation time. You can defined more final static boolean to enable/disable insertion of trace specific code with a better “grain”.

Next you can begin to create you logger. These loggers are static and final. You can create one by calling the static method *createLogger(String name, Object class, String[] description)* of the class *LoggerProviderSingleton*.

Finally, in the code, you can use one of the loggers defined by getting its reference. And then call one of the log methods:

- log(BasicLevel , Object msg)
- log(BasicLevel , Object msg , Throwable t)
- log(BasicLevel , Object msg , Object msg , Object msg)
- log(BasicLevel , Object msg , Throwable t , Object msg , Object msg)

Example of MonologTopics Class :

```
Package org.omg.ccm.example;

import org.objectweb.util.monolog.api.Logger;
import org.objectweb.util.monolog.provider.LoggerProviderSingleton;

final public class MonologTopics{

      final static Boolean enableTrace = true ;
      // false; // enable/disable trace at compilation time

      final static Logger concernLogger =
LoggerProviderSingleton.creatLogger(
             "exemple.preoccupation",
            innerClass.class,
            new String[] {"description 1"," description 2"});
            }
}
```

Where *innerClass* is a class of the package *org.omg.ccm.example*.

Use of *concernLogger* in the class *innerClass*:

- import:

```
org.objectweb.util.monolog.api.Logger
org.objectweb.util.monolog.api.BasicLevel
```

- Direct uses :

```
if (MonologTopics.enableTrace)
      MonologTopics.concernLogger.log(BasicLevel.DEBUG, "message");
```

- Get a reference of *concernLogger* and use it to call *Logger.log* methods:

```
if (MonologTopics.enableTrace)
      Logger logger_ = MonologTopics.concernLogger;
if (MonologTopics.enableTrace)
logger_.log(BasicLevel.DEBUG, "message");
```

6.2.3 Monolog Configuration

The loggers configuration and their output Appenders are configured in Monolog factory. This factory is configured by a Java properties file (in the future by a XML file).

The Monolog configuration file for OpenCCM is named *monolog.CCM.properties* It is located in each demo directory.

Syntax of the monolog property configuration file :

- Handlers : An handler represents an output. Monolog provides three standard handlers(console, file and RollingFile) and a generic handler which permits to configure any handler.

  An Handler is identified by its name. It has a type and few others properties. To define an handler is needed to give its name and its type. The handler definition is composed by several lines where each line matches to a property. The general expression is the following:

  handler.<handler name>.<property name> <property value>

  Here is an example of handler definition:

```
handler.my_console_output.type Console
handler.my_console_output.output System.out
handler.my_console_output.pattern %m%n
```

  This code defines an handler which prints the messages to the console. The output is the standard output (not the error stream). Finally the pattern (the format of the messages) is very simple: one message by line. For more details about pattern see the pattern section

- Loggers : A Logger is identified by names. However we consider that each logger has a main name. This name is used to identify it in the property file. There are several configurable things on a Logger instance.
  - The first configurable element is its level:

    logger.<dotted logger name>.level <level value>

    The <dotted logger name> part represents the main name of the logger. This string can be composed of dot since the names can describe a hierarchy. The <level value> part represents the value of the level logger. This value MUST be a level name. The level name can represent a predefined level of an intermediate level already defined in the LevelFactory. Here are some examples of logger level definition:

```
logger.org.objecweb.foo.level DEBUG
```

```
logger.org.objecweb.foo.level my_level4
logger.root.level WARN
```

IMPORTANT: The last example shows the way to configure the root logger instance. "root" is the particular name which designs this logger.

- The second configurable element is the list of additional topics. The general expression is the following:

  logger.<dotted logger name>.topic.<topic id> <additional topic>

  The <topic id> is a simple integer value which permits to make the difference between each property. Here are some examples of logger topic definition:

```
logger.org.objecweb.foo.topic.0 com.bar
logger.org.objecweb.foo.topic.1 fr.bar
logger.a.b.topic.0 e
```

- The third configurable element is the handler list. The general expression is the following:

  logger.<dotted logger name>.handler.<handler id> <handler name>

  The <handler id> is a simple integer value which permits to make the difference between each property. Here are some examples of logger handler definition:

```
logger.org.objecweb.foo.handler.0 my_console_output
logger.org.objecweb.foo.handler.1 my_file_Log.txt
logger.a.b.handler.0 my_console_output
```

- The fourth configurable element is the handler inheritance. It is possible to indicate if a logger inherits handlers of its ancestor. The property is named additivity and the possible values are true or false. The default value is true, i.e. by default the logger inherits handlers of their ancestors.

```
logger.org.objecweb.foo.additivity false
```

6.2.4 Local TraceService Class diagram

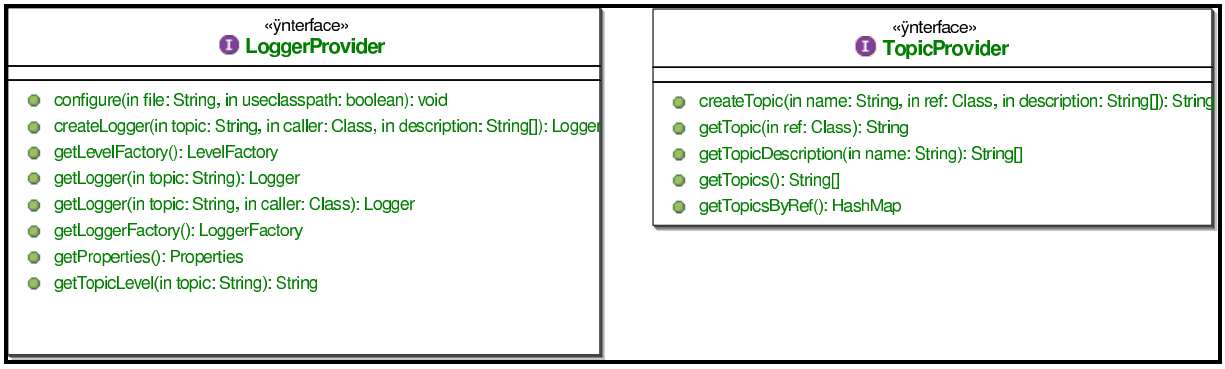

Figure 67 - org.objectweb.util.monolog.provider.api

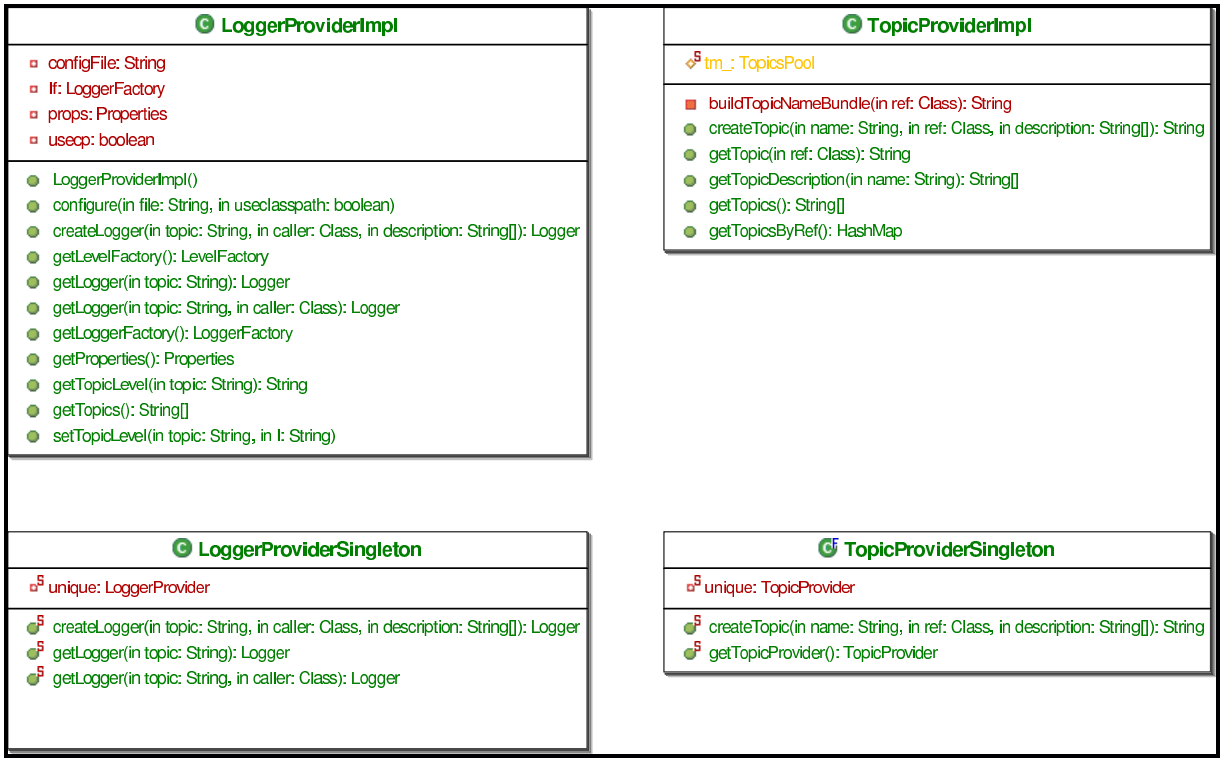

Figure 66 - org.objectweb.util.monolog.provider.lib

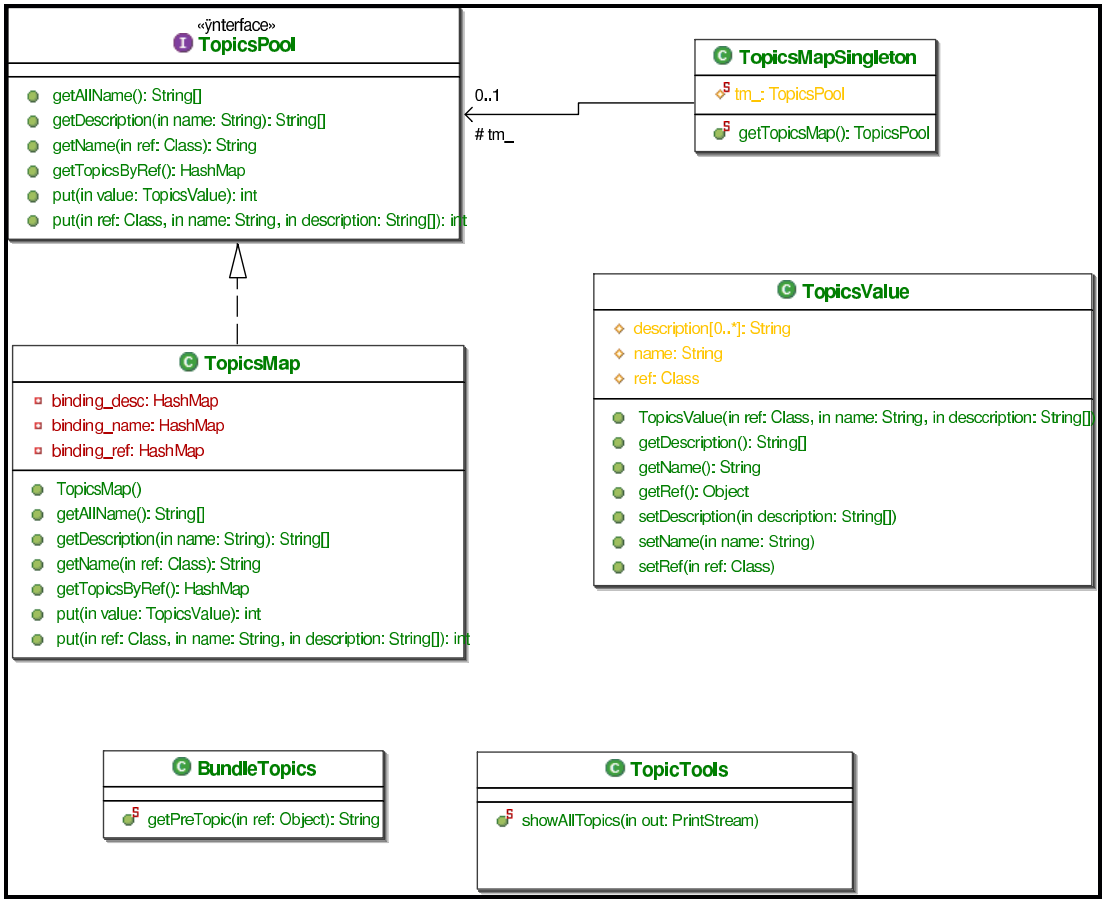

Figure 68 - org.objectweb.util.monolog.provider.lib.topics

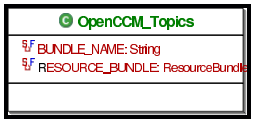

Figure 68 - org.objectweb.ccm.trace

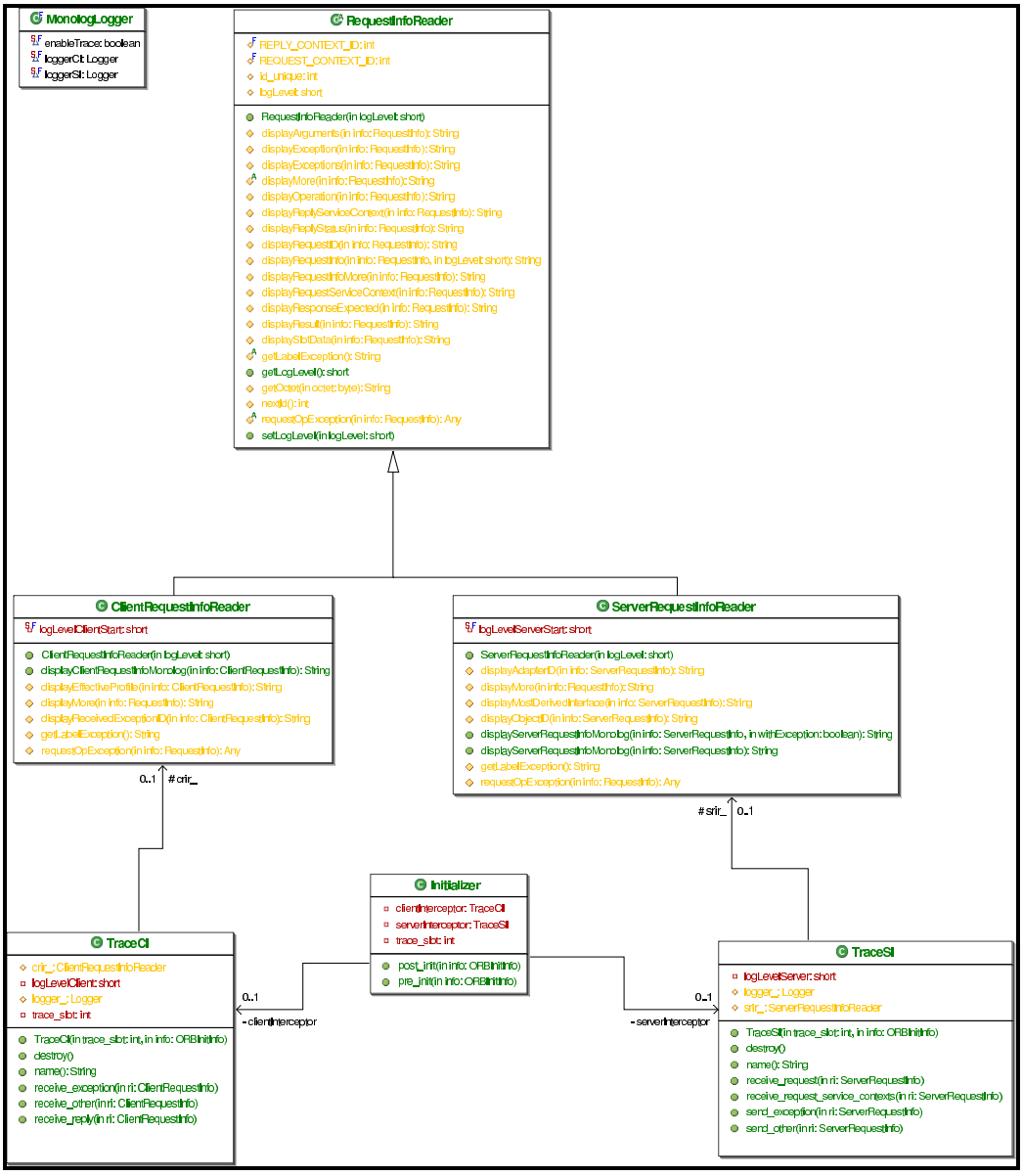

Figure 69 – The org.objectweb.corba.trace.PI package

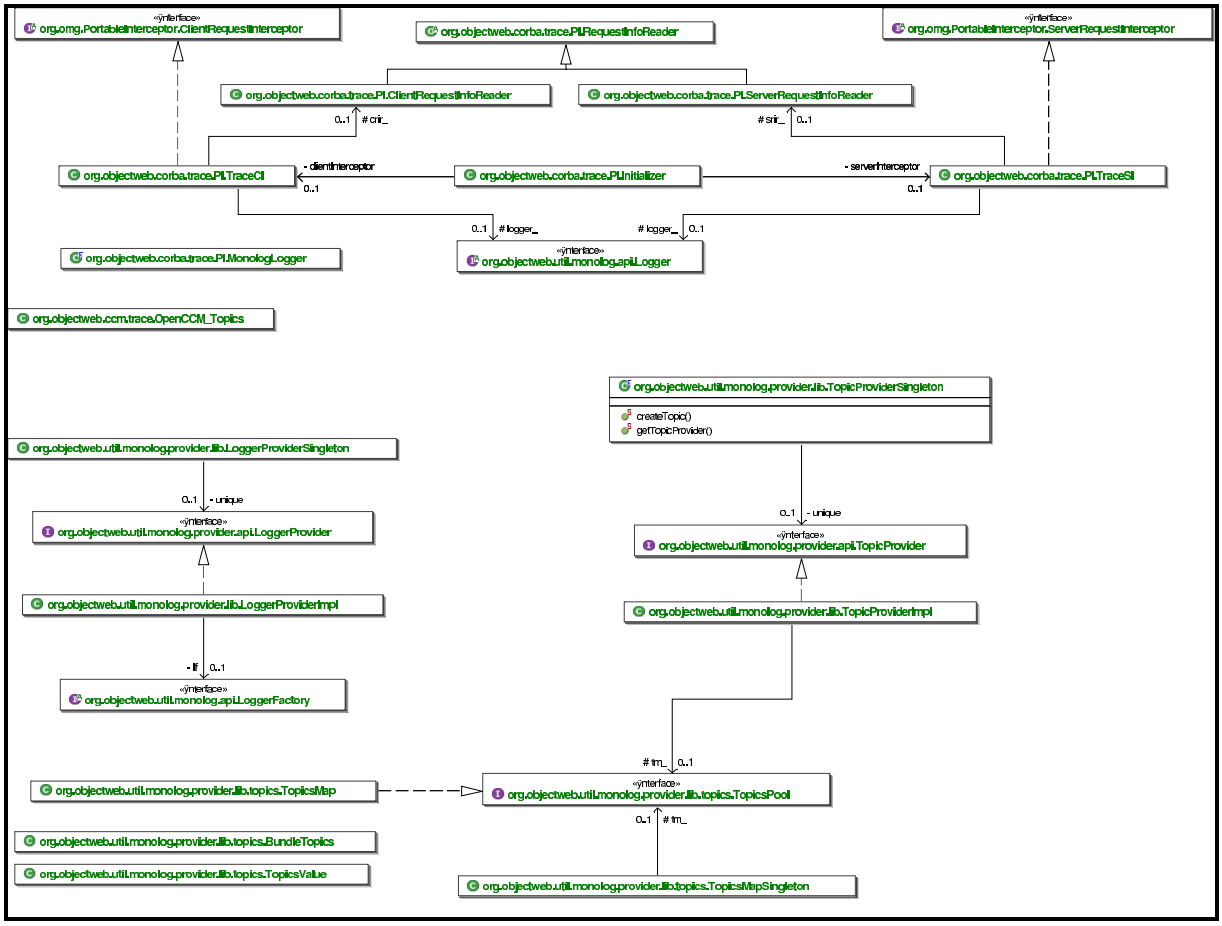

Figure 70 – Overview of the Trace Service