ARTIST

FP7-317859

Advanced software-based seRvice provisioning and migraTIon of legacy Software

D9.7

Integrated Environment for maintaining / developing forward engineering process

Editor(s):	Alexander Bergmayr Manuel Wimmer Javier Troya
Responsible Partner:	TUWIEN
Status-Version:	Final – V1.0
Date:	31/03/2015
Distribution level (CO, PU):	PU

Project Number:	FP7-317859
Project Title:	ARTIST

Title of Deliverable:	Integrated Environment for maintaining / developing forward engineering process
Due Date of Delivery to the EC:	31/03/2015

Workpackage responsible for the Deliverable:	WP9
Editor(s):	Alexander Bergmayr (TUWien) Manuel Wimmer (TUWien) Javier Troya (TUWien)
Contributor(s):	ATOS, Tecnalia, Sparx, INRIA, Engineering
Reviewer(s):	Leire Orue-Echevarria (TECNALIA)
Approved by:	All Partners
Recommended/mandatory readers:	WP 6, WP 7, WP 8, WP 11

Abstract:	This deliverable comprises automatically executable transformations needed for deploying the modernized applications in specific Cloud infrastructures
Keyword List:	Transformation Composition, Transformation Configuration, Transformation Parameterization, Repository
Licensing information:	Generally EPL (open source), indicated otherwise. The document itself is delivered as a description for the European Commission about the released software, so it is not public.

Document Description

Document Revision History

Version	*Date*	*Modifications Introduced*	
		Modification Reason	*Modified by*
v0.1	25/02/2015	Table of Contents	TUWIEN
v0.5	13/03/2015	Content for all Sections	ATOS, ENGINEERING, TECNALIA, TUWIEN
v0.9	17/03/2015	Compilation for internal review	TUWIEN
v1.0	27/03/2015	Final version	ATOS, ENGINEERING, TECNALIA, TUWIEN

Table of Contents

Table of Figures

Terms and abbreviations

EC	European Commission
MDFE	Model Driven Forward Engineering
API	Application Programming Interface
GML	Graphical Modelling Language
SOTA	State Of The Art
MT	Model Transformation
M2M	Model to Model
ATL	Atlas Transformation Language
M2T	Model to Text
PIM	Platform Independent Model
MUT	Model Understanding Toolbox
PDM	Platform Dependent Model
DSL	Domain-Specific Language
OCL	Object Constraint Language
MDE	Model-Driven Engineering

Executive Summary

The ARTIST project aims at providing concepts, techniques, and tools for the modernization of software by a migration to the cloud. Many tools have been developed in the context of different WP's of the ARTIST project. Allowing software engineers and modellers to apply these tools for real world migrations scenarios, they need to be integrated into a common environment. In this deliverable, we report on the ARTIST integrated environment that provides a common interface to the ARTIST tools. In this sense, the integrated environment complements the methodology process tool, cf. WP6, with dedicated tool-support for software engineers and modellers to carry out a migration process.

As in ARTIST we work towards a model-based engineering approach to support cloud-oriented software migration, model transformations play a key role in the transition of software to the cloud. In fact, such a migration often requires carrying out several different transformations to ultimately gain the required software artefacts. These transformations need to be properly chained and maintained because typically a diverse set of transformations realized by different transformation languages and technologies need to be executed in an appropriate order. For this reason, we have developed a dedicated language that allows software engineers and modellers to compose multiple transformations into a chain, thereby enabling the automatic execution of such transformations if the chain becomes executed. The language supports transformation technologies mainly applied in the context of reverse-engineering application code into models, cf. WP8, and forward-engineering application code hosted by a cloud environment from models, cf. WP9. In this deliverable, we give an overview of the key concepts constituting this language and show how it can be applied for chaining transformations that have been developed in the context of the ARTIST project for supporting cloud-oriented software migration.

Moreover, transformations developed in the context of WP9 mainly realize model-to-model transformations with the purpose of refining and optimizing them towards a selected cloud environment and producing the pertinent application code from such models by relying on model-to-code transformations. In this deliverable, we report on a selected set of patterns for which we have developed concrete transformations and profiles, aiming to optimize certain software artefacts for a selected cloud environment. In this respect, the quality of reverse-engineered models plays a crucial role as they are considered as starting point for applying such transformations in the forward engineering phase, cf D8.3. Only if they their quality is appropriate, the quality of the models and generated application code as a result of possibly chained transformations will be sufficient and useful for software engineers and modellers.

The ARTIST migration process provides techniques to investigate on the feasibility of a migration to the cloud mainly by providing metrics, including source-code maintainability. In this deliverable, we report on the calculation of the maintainability metric of application code generated as a result of an ARTIST migration scenario with purpose to reason about possible improvements.

1 Introduction

1.1 About this deliverable

The integration of the ARTIST tools developed in different WPs into a common environment aims at improving their application in real-world cloud migration scenarios including the ARTIST use cases. In this deliverable, we report on the ARTIST integrated environment that provides a common interface to all integrated ARTIST tools on top of the Eclipse environment. It mainly targets software engineers and modellers carrying out migration scenarios to the cloud. As in ARTIST we work towards a model-based engineering approach to support such a software migration, model transformations play a key role in the transition of software to the cloud. In fact, a migration scenario often requires carrying out several different transformations to ultimately gain the required software artefacts. These transformations need to be properly chained and maintained because typically a diverse set of transformations realized by different transformation languages and technologies need to be executed in an appropriate order. For this reason, we have developed a dedicated language that allows multiple transformations to be composed into a transformation chain, thereby enabling their automatic execution if the chain becomes executed. In this deliverable, we give an overview of the key concepts constituting this language and show how it can be applied for chaining transformations that have been developed in the context of the ARTIST project for supporting cloud-oriented software migration. Moreover, in the ARTIST project transformation techniques have been applied in combination with annotation-based modeling [4] for realizing optimization patterns as reported by WP9 Deliverables. In this deliverable, we report on a selected set of patterns for which we have developed concrete transformations and profiles, aiming to optimize certain software artefacts for a selected cloud environment. In this respect, not only model-to-model transformations are applied but also model-to-code transformations that generate cloud-specific application code. As the ARTIST migration methodology provides, among others, techniques to investigate on the feasibility of a migration to cloud mainly by providing metrics, including source-code maintainability [2]. In this deliverable, we report on the calculation of the maintainability metric of application code generated as a result of an ARTIST migration scenario with purpose to reason about possible improvements.

1.2 Fitting into the overall ARTIST solution

The ARTIST integrated environment supports the ARTIST migration methodology from a technical perspective by providing a tool suite to carry out real-world migration scenarios including the ARTIST use cases. In this sense, it complements the methodology process tool with a dedicated model-based tool-support for software engineers and modellers to carry out such a process. As transformations play a crucial role in such a migration scenario, the transformation composition language allows software engineers and modellers to chain them into a coarse-grained transformation. This language supports transformation technologies mainly applied in the context reverse-engineering application code into models, cf. WP8, and forward-engineering application code hosted by a cloud environment from models, cf. WP9. Considering the latter, transformations developed in the context of WP9 mainly realize model-to-model transformations with the purpose of refining and optimizing them towards a selected cloud environment and producing the pertinent application code from such models. Clearly, the quality of reverse-engineered models plays a crucial role as they are considered as starting point for forward-engineering activities. Only if they their quality is appropriate, the quality of the models and generated application code as a result of such a transformation chain will be sufficient and useful for software engineers and modellers. Measuring the quality of generated application code is provided by calculating dedicated metrics including maintainability.

1.3 Main Innovations

In this deliverable, we set the focus primarily on (i) the integration of ARTIST tools to support model-based software migration to the cloud, (ii) the development of a composition language to chain transformations as part of the concrete cloud-oriented migration scenario, (iii) advances in the development of transformations and profiles to realize cloud optimization patterns for a selected environment, and (iv) advances in the calculation of automatically generated source-code metrics.

- **ARTIST integrated environment.** Provides a common interface to ARTIST tools to support software engineers and modellers to carry out software migrations to the cloud.
- **Transformation composition language.** Allows multiple transformations to be composed into a transformation chain, thereby enabling their automatic execution if the chain becomes executed and the reuse of them in the sense that they are loosely coupled by a more coarse-grained transformation.
- **Transformations and profiles for cloud-based optimization.** Supports the improvement of software artefacts with novel cloud optimization opportunities.
- **Quality measurements for generated application code.** Makes the generated code of ARTIST tools measureable.

1.4 Delivered Components

All realized components are packaged either as Eclipse plugins or Eclipse projects. The selected license is the Eclipse Public License (EPL)[1] which is a known as a “commercial-friendly” open source license. This should facilitate the future potential reuse and integration of the toolbox (or at least of some of its components) by external partners.

The source of the integrated environment and the transformation composition language is located in the public ARTIST repository:

```
https://github.com/artist-
project/ARTIST/tree/master/source/Tooling/migration/integrated environment
```

Regarding advances in cloud optimization pattern, modifications have been carried out to the model cloudification framework (MCF) as introduced in D9.3 and D9.5:

```
https://github.com/artist-
project/ARTIST/tree/master/source/Tooling/migration/modernization/MCF
```

Regarding advances in measusring the qualitiy of generated source code, the respective components are located at:

```
https://github.com/artist-project/ARTIST/tree/master/source/Tooling/pre-
migration/technical feasibility tool/SCC
```

1.5 Document structure

The document is structured as follows. In Section 2, we introduce the ARTIST integrated environment on top of the Eclipse environment. We introduce the transformation composition language in Section 3. In Section 4 we report on the realization of a set of optimization patterns for a selected cloud environment, while in Section 5, we report on how the

[1] „Eclipse Public License,“ [Online]. Available: http://www.eclipse.org/legal/epl-v10.html.

maintainability metric can be calculated for application code generated as a result of an ARTIST migration scenario before we conclude in Section 6.

2 ARTIST Integrated Environment

2.1 Functional Description

ARTIST Suite consists of a bundle of different tools supporting the ARTIST Cloud migration methodology. As these tools support different phases and tasks within the methodology, and target different roles (and therefore, providing different interfaces for those users), they can initially be not sufficiently integrated from different perspectives: usability, interoperability, seamlessly integration along the ARTIST methodology and so on. The fact that the different ARTIST tools have been developed according to different functional constraints, technical paradigms or different architecture platforms - because of technical considerations or constraints – imposes some challenges into their mutual integration. To improve the user perception of the ARTIST Suite, as a single seamlessly integrated tool, different technical initiatives have been conducted by ARTIST in different work packages:

- The ARTIST methodology provides the conceptual and behavioural integrated process that drives the operation of the entire ARTIST Suite and its constituting tools.
- The Methodology Process Tool (MPT) provides another mechanism to improve the behavioural and functional integration of the ARTIST Suite tools, which are orchestrated within the MPT, in the different walk-through guides provided (e.g. the Eclipse Cheat Sheets), following the personalised ARTIST methodology.
- The ARTIST Integrated Environment, described in this section, provides a seamlessly integrated common user interface for all those ARTIST Suite tools that run within the Eclipse environment.

The ARTIST Integrated Environment is an Integrated Development Environment (IDE), hosted in the Eclipse IDE, which offers the Model-Driven-Engineering tools developed by ARTIST, which are compatible with Eclipse. These ARTIST tools, mostly targeting software engineers and modellers, cover a significant percentage of all the ARTIST tools. Nonetheless, few ARTIST tools cannot be directly integrated, simply because they are not compatible with the ARTIST framework. Nonetheless, for these cases, other loose coupling integration approaches have been addressed as well, cf. D6.4.1. After this conceptual introduction of the scope and motivation of the ARTIST Integrated Environment, we continue describing the main functionality it offers to the end-user:

- A single delivery and installation procedure, whereby the entire ARTIST Integrated Environment can be installed, either from a distributable bundle, or from the ARTIST update site. Users can install the complete ARTIST Suite or individual components (e.g. Eclipse features).
- An integrated configuration for all ARTIST Suite tools, within a single tree-structure in the ARTIST preference pages. Additionally, endpoints to access external (i.e. Web based) ARTIST tools can be configured in these pages.
- Top tool bars and menu bars to access external Web-based ARTIST Tools, such as the Maturity Assessment Tool, the Methodology Process Tool, the Certification Tool, etc. as well as wizard based tools of the ARTIST Suite, such as the Enterprise Architect Bridge, the Reusability Trace Tool, etc. These external Web-based tools can be accessed through the internal Eclipse Web browser, or through an external one, depending on the configuration.
- A Wizard to create an ARTIST Cloud migration project, based on Java, which defines a Cloud migration project folder structure according to the ARTIST MDE methodology.

- A layout organization of ARTIST tool views in Eclipse perspectives. The main ARTIST perspective includes by default the Repository view, from where the user can browse and access reusable artefacts available in the repository.
- Homogeneous usability and access mechanism (through the user interface) to the Eclipse-based ARTIST tools, through contextual menus that offer context-sensitive options based on selected artefacts in the local workspace. This common access mechanism complements other tool-specific ones, already provided by the tools themselves, such as the Eclipse runtime configurations.
- Grouping of the ARTIST tools within common toolsets: Model Discovery Toolset, Model Understanding Toolset, Model Cloudification and Optimization Toolset or Code Generation toolset.

These user interface features are aiming to improve the interoperability and usability of the ARSTIST tools within the Eclipse environment, and supporting the access to external ARTIST Web-based tools as well, were agreed by the consortium (both technical and use case providers) in an integration workshop held during an ARTIST general assembly.

2.2 Technical Description

ARTIST Integrated Environment (ARTIST IDE) is implemented using Eclipse PDE (Plug-in Development Environment) collecting most of the ARTIST tools as plugins and features within its body allowing for a single installation for ARTIST tools and also an update site. ARTIST IDE also owns a parent plugin as the main feature, which is depending on by several tools, defining an Eclipse perspective for ARTIST, a preference page tree entry housing preferences for some of the tools and an ARTIST top menu and a toolbar for launching external tools. Next figure abstractly outlines the ARTIST IDE's architecture.

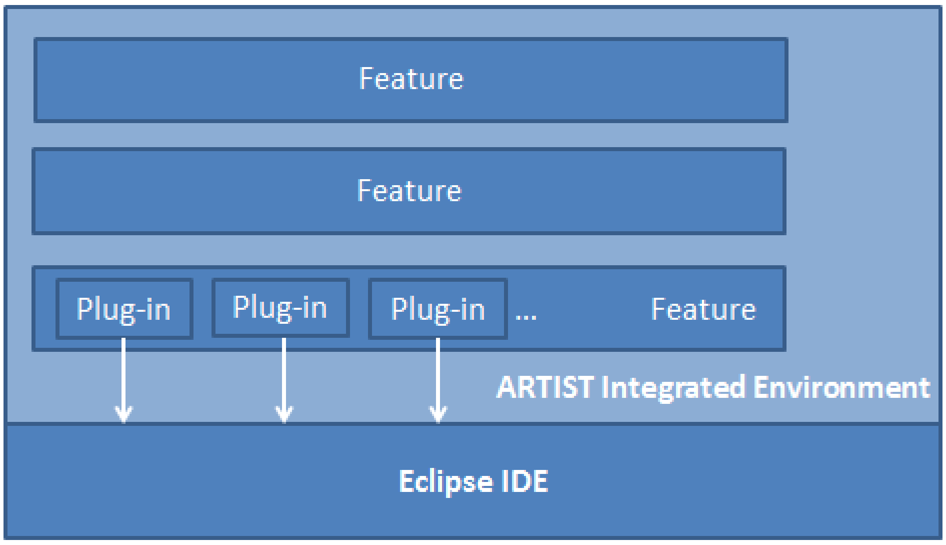


Figure 1 ARTIST Integrated Environment

2.2.1 ARTIST Suite

ARTIST Perspective: contributing to Eclipse's perspectives extension by defining a new perspective within Eclipse specialized for ARTIST. This perspective layout contains the Repository View, Problems View and the Package Explorer as default. ARTIST Perspective is created by adding an extension for *org.eclipse.ui.perspectives* extension point in the plugin.xml and also defining a class implementing the *IPerspectiveFactory* interface of Eclipse UI package to create the initial layout.

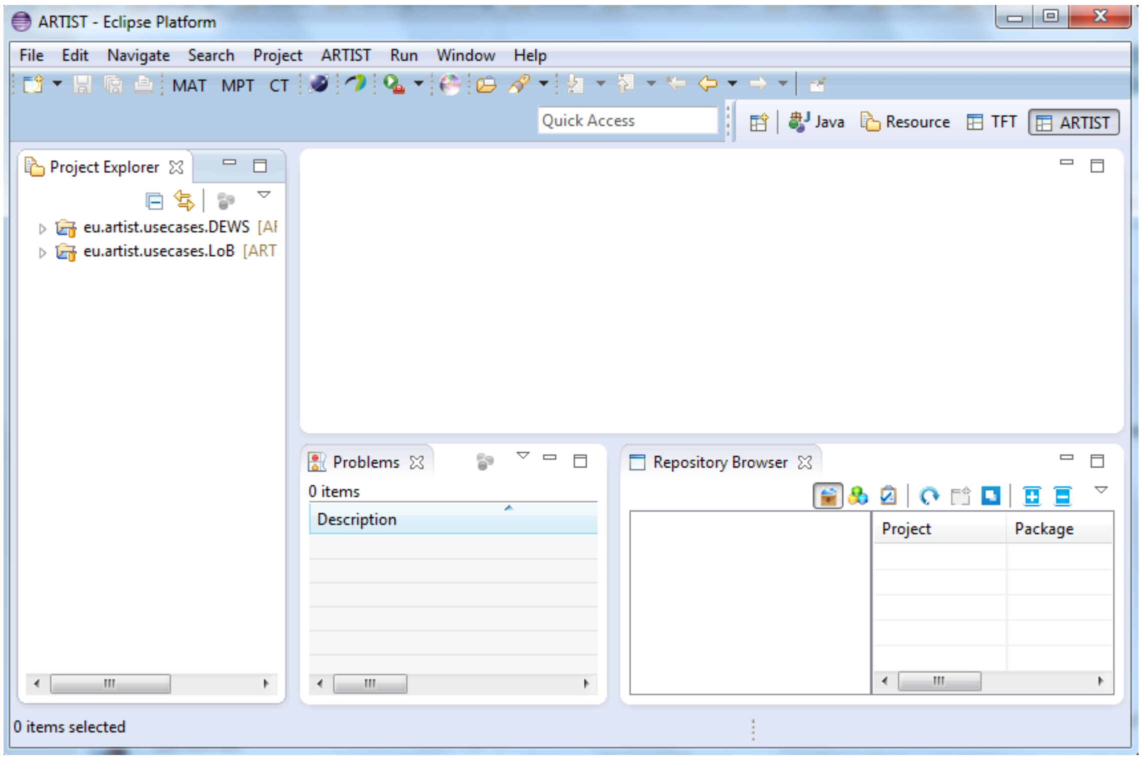
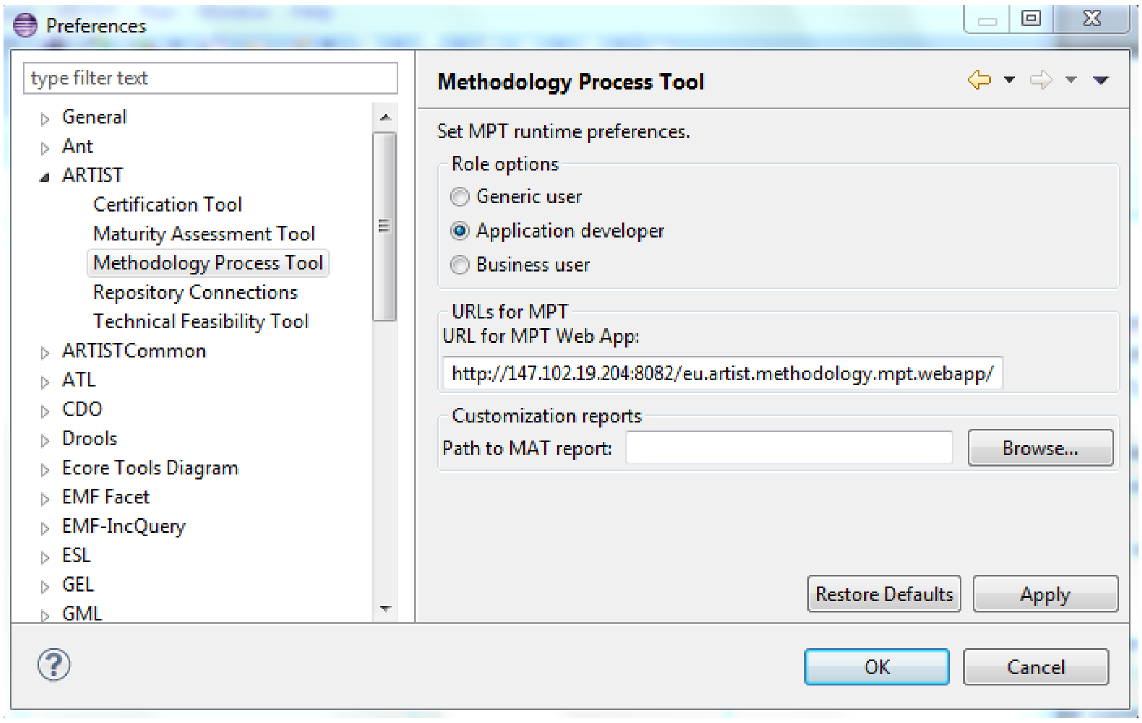


Figure 2 ARTIST Perspective

ARTIST Preference Pages: contributing to Eclipse's Preferences and PreferencePages extensions by defining dedicated preference pages.

- ARTIST page contains a single checkbox which offers the user the option to launch the external tools whether in Eclipse's internal web browser or the system browser.
- Technical Feasibility Tool page too contains a single checkbox to let the user set a default path for the TFT Report generation path.
- Methodology Process Tool page consists of a radio button group by which the user can select a user role for MPT, a text field for storing the default URL for MPT Web App and again a text field to store the path value for the MAT report to be consumed by MPT cheat sheets.
- Repository Connections page contains an SWT table to collect and store connection information (name and URL) for the ARTIST Repository.
- Maturity Assessment Tool page allows the user to set the default MAT URL on the text field provided and also user credentials for MAT.
- Certification Tool page contains a text field to store the default value for the Certification Tool's URL.

All defined pages created by adding an extension for *org.eclipse.ui.preferencePages* extension point in the plugin.xml and also defining a class implementing the *IWorkbenchPreferencePage* interface of the Eclipse's UI package. The preference page classes in ARTIST Suite also extend the *FieldEditorPreferencePage* class of Eclipse UI for the ease of layout creation with regards to coding.



Methodology Process Tool

Set MPT runtime preferences.

Role options
( ) Generic user
[✓ Application developer]
( ) Business user

URLs for MPT
URL for MPT Web App: http://147.102.19.204:8082/eu.artist.methodology.mpt.webapp/

Customization reports
Path to MAT report:

Figure 3 ARTIST preference pages

ARTIST Top Menu and Toolbar: Menu and the toolbar actions allows the user to create an ARTIST migration project and launch external tools defined in the context of ARTIST; Certification Tool, Maturity Assessment Tool and Methodology Process Tool.

Both the ARTIST top menu and toolbar is created by adding an extension for *org.eclipse.ui.menus* extension point in the plugin.xml by adding a menu contribution. The menu actions invoke an Eclipse command, defined in the plugin.xml by extending the *org.eclipse.ui.commands* extension point. These actions send a parameter to the command handler class which is then interpreted and the appropriate menu action is invoked, be it the external tool or the ARTIST migration project wizard.

2.2.2 Model Discovery Toolbox

Model Discovery Toolbox feature consists of several plugins, which together forms the tools in this toolbox. Currently, this toolbox, reported in WP8 Deliverables, consists of the tools; Java2Model, Java2UMLClass, Java2UMLActivity, XMLAggregator, SQL Model Discoverer and JUMP. All these tools except JUMP contribute to the ARTIST Integrated Environment by adding menu actions to Modisco's Discovery context menu item. This is managed by adding an extension for *org.eclipse.ui.menus* extension point in the plugin.xml using the menu category ID defined by Modisco.

JUMP tool on the other hand, defines a Run Configuration which can be accessed by opening the Eclipse's default Run Configuration View. This contribution is extended by adding a context menu option to projects which have Java nature. The menu option allows the user to create and open a JUMP run configuration with the properties of the selected Java Project. This is also managed by adding an extension for *org.eclipse.ui.menus* extension point in the plugin.xml that is linked to a command invoking the run configuration of the JUMP tool.

The user manuals and figures further explaining the tools can be found in the tool's respective deliverables (WP8 series of deliverables).

2.2.3 Model Understanding Toolbox

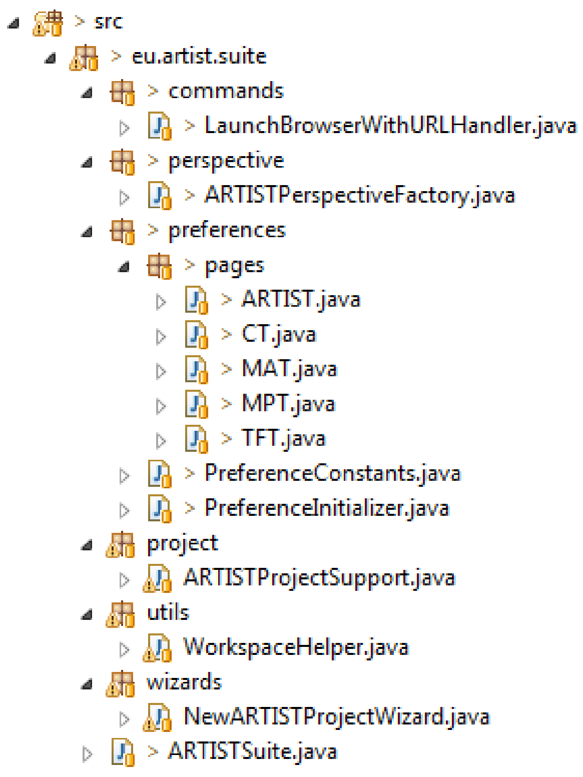
The tools in Model Understanding Toolbox are; Component Model Generator, PIM Abstractor, Annotation Based Slicing and CMM (Computation of Model Metrics). All of these tools contribute to the Eclipse's menu extension by defining menu actions under the Model Understanding Toolbox context menu for UML files. PIM abstractor and Component Model Generator also assist the user with a dialog on which the user can make a selection among the pre-defined abstractor types and UML profiles to be used in the model abstraction and generation processes. This dialogs are basically implemented as Java classes extending the *TitleAreaDialog* of Eclipse's JFace API. The user manuals and figures further explaining the tools can be found in the tool's respective deliverables (WP8 series of deliverables).

2.2.4 Code Generation Toolbox

This toolbox feature is formed of Java Code Generator and Maintainability Metric Calculator (MMC) tools, both of which define extensions to Eclipse menus by action additions to context menus of UML files. These actions are grouped under the Code Generation Toolbox menu item. MMC also has a custom dialog which helps the user in selecting a source code location of the selected UML model. The dialogs and menu entries for this feature are implemented the same way as the Model Understanding Toolbox's menu extensions and dialogs. The user manuals and figures further explaining the tools can be found in the tool's respective deliverables (WP8 series of deliverables).

2.2.5 Package information

Since all ARTIST tools' individual package information are within their respective deliverables, this section will only describe the delivery package of the main feature (ARTIST Suite), which consists of the folder structure shown below.



```
> src
  > eu.artist.suite
    > commands
      > LaunchBrowserWithURLHandler.java
    > perspective
      > ARTISTPerspectiveFactory.java
    > preferences
      > pages
        > ARTIST.java
        > CT.java
        > MAT.java
        > MPT.java
        > TFT.java
      > PreferenceConstants.java
      > PreferenceInitializer.java
    project
      ARTISTProjectSupport.java
    utils
      WorkspaceHelper.java
    wizards
      NewARTISTProjectWizard.java
    > ARTISTSuite.java
```

Figure 4 Package structure of ARTIST Suite

eu.artist.suite.commands: Contains the *LaunchBrowserWithURLHandler.java* which handles the browser launch actions of external tools.

eu.artist.suite.perspective: Contains the *ARTISTPerspectiveFactory.java*. This class is responsible for the creation of the main layout of the ARTIST perspective.

eu.artist.suite.preferences: Contains the *PreferenceConstants.java* which holds constant and default values that are being used by the preference pages and *PreferenceInitializer.java* which initializes the preference values whenever a preference page is opened.

eu.artist.suite.preferences.pages: This package contains the implementations of all the preference pages defined within the context of ARTIST. Each class in this package defines a unique page in the Eclipse preferences tree view.

eu.artist.suite.project: Contains the *ARTISTProjectSupport.java*. This class offers a method for creation of an ARTIST Migration Project.

eu.artist.suite.utils: The package contains a single class called *WorkspaceHelper.java*. This class offers methods for creating a base project, adding nature to a project, creation of folders and setting classpath for library files.

eu.artist.suite.wizards: Contains the *NewARTISTProjectWizard.java* which is an extension of the Eclipse wizard, enabling creation of an ARTIST project using the Eclipse wizards.

2.3 User Manual

ARTIST Integrated Environment and all its features plugin will be available for installation using Eclipse's Install New Software feature via ARTIST's update site. Now, it can be downloaded and installed manually. Eclipse 4.2 (Juno) is required for installation but Eclipse 4.3 (Kepler) is recommended.

Installation Procedure of Required Plugins

ATL plugins are required to be installed. ATL plugins may be installed by following the installation procedure below:

1. On the Help menu of Eclipse, click Install New Software.
2. From the "Work with" dropdown list select Kepler or Juno depending on your Eclipse version. If you do not have these entries in the list continue on from step 2a otherwise continue from step 3.
    a. Click Add button at the right side of the Work with dropdown list.
    b. Write a name of your choice for the Kepler/Juno repo.
    c. Write http.download.eclipse.org/releases/kepler or http://download.eclipse.org/releases/juno to the Location field and click Ok.
3. Select ATL SDK, click next and follow the instructions on screen.
4. Restart Eclipse.

Installation of ARTIST Integrated Environment

The ARTIST IDE update site will be published at the ARTIST website www.artist-project.eu available for installation using Eclipse's *Install New Software* feature. Currently, and anytime, the user can download the update site zip file from https://github.com/artist-project/ARTIST/tree/master/binary/ARTIST_IDE and follow the instructions below for installation.

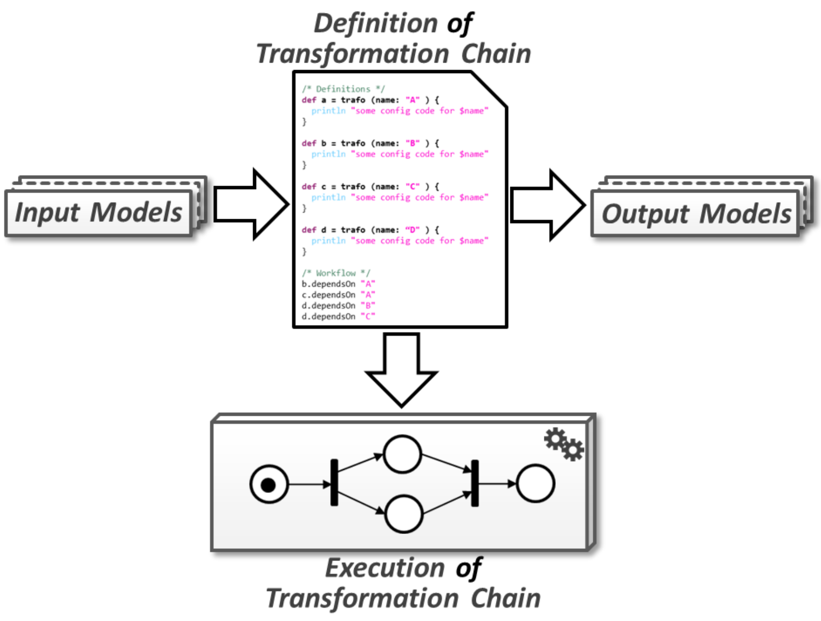
1. Download the update site zip file from: https://github.com/artist-project/ARTIST/tree/master/binary/ARTIST_IDE
2. Extract the Zip contents to a desired location.
3. On the Help menu of Eclipse, click Install New Software.
4. Click the Add button at the right side of the Work with dropdown list.
    a. Click the Local button and select the update site folder extracted from the zip file on step 2. Click Ok button.
    b. Click Ok button.
5. Select the ARTIST features you want to install and follow the instructions on screen.
6. Restart Eclipse.

3 Transformation Composition Language

A modernization scenario often requires carrying out several different transformations to ultimately gain the modernized software artefacts. These transformations need to be properly chained and maintained because typically a diverse set of transformations realized by different transformation languages need to be executed in an appropriate order in particular if there are dependencies between these transformations. For instance, a transformation may require models as input that is produced by another transformation as output. In such a case, the order in which they are executed is vital for the produced result.

3.1 Functional Description

We have developed a dedicated language that allows multiple transformations to be composed into a transformation chain, thereby enabling their automatic execution if the chain becomes executed. Chaining transformations fosters reuse in the sense that existing transformations are loosely coupled by a more coarse-grained transformation [1]. Such a loose coupling ensures that the developers still have the control over the execution of the single transformations. The latter is important not only to pause and resume a transformation chain in case of developers need to intervene, e.g., manual changes need to be carried out on produced models, but also to partially re-execute the transformation chain as a result of updates to one or more transformations. Such an incremental transformation execution can significantly reduce the runtime overhead particularly when computation-intensive transformations are marginally revised [2]. Moreover, it can ensure that manual changes to existing models prior the re-execution of a transformation in the chain are retained because they are only updated instead of entirely re-created.


Definition of Transformation Chain
Input Models
Output Models
Execution of Transformation Chain

```
/* Definitions */
def a = trafo (name: "A" ) {
  println "some config code for $name"
}

def b = trafo (name: "B" ) {
  println "some config code for $name"
}

def c = trafo (name: "C" ) {
  println "some config code for $name"
}

def d = trafo (name: "D" ) {
  println "some config code for $name"
}

/* Workflow */
b.dependsOn "A"
c.dependsOn "A"
d.dependsOn "B"
d.dependsOn "C"
```

Figure 5 Conceptual overview of transformation composition

From a conceptual perspective, a transformation chain is considered as a single transformation, where its input is determined by the entry point of the transformation chain, i.e., transformations that do not require input from other transformations, and the end point of the transformation chain determines its output. The latter is produced by transformations on which no other transformations depend. As a result, a transformation chain is a directed graph, where the arcs in the graph represent a dependency of one transformation to another of the chain. We exploit a place/transition Petri net [3] as the formal representation to encode transformation chains and its well-defined operational semantics to execute them. In doing so, the analysis techniques supported by this formalism can directly be used to, e.g., check the reachability of a certain transformation from another one in the chain. Moreover, the fact that Petri nets permit concurrent state transitions enables transformations to be executed in parallel. Finally, because the placement of the tokens captures the state of the chain's execution, it allows developers to intervene the chain, thereby both a black-box as well as a white-box view on a transformation chain is provided. Figure 5 gives a conceptual overview of defining and executing a transformation chain.

The definition of a transformation chain is enabled by a Groovy[2]-based domain-specific language, which is inspired by current build tools such as Gradle[3]. The following Listing shows the definition of a transformation chain consisting of five transformations that depend on each other. The first transformation produces a Java model from a Java code base. In fact, MoDisco is used to perform this first transformation, which is realized as a model discoverer in the context of WP8. In a second step, a UML model is produced from the previously obtained Java model, where this UML model is considered as input for a slicing transformation in the third step. Then, in the fourth step the sliced UML model is refined towards the Google App Engine before in a final step code is generated from it. All the transformations in this scenario have been developed in the context of WP8 and WP9.

```
def java2javamodel = trafo ( name: "java2javamodel", of: "MoDisco" ) {
      project = "JavaProject"
}

def java2uml = trafo ( name: "java2uml", of: "ATL" ) {
      metaModel "Java", "./Java.ecore"
      metaModel "UML", "./UML.ecore"

      input "Java", "IN", "./javamodel.xmi", "JavaSource"
      output "UML", "OUT", "./umlmodel", "Table"

      transformation = "./Java2UML.asm"
}

def uml2sliceduml = trafo ( name: "uml2sliceduml", of: "ATL/EMFTVM" ) {
      metaModel "UML", "./UML.ecore"

      input "Java", "IN", "./javamodel.xmi", "JavaSource"
      output "UML", "OUT", "./umlmodel", "Table"

      transformation = "./UML2SlicedUML.asm"
}

def uml2refineduml = trafo ( name: "uml2refineduml", of: "ATL/EMFTVM" ) {
      metaModel "UML", "./UML.ecore"

      input "Java", "IN", "./javamodel.xmi", "JavaSource"
      output "UML", "OUT", "./umlmodel", "Table"

      transformation = "./UML2SlicedUML.asm"
}
```

[2] Groovy: http://groovy-lang.org
[3] Gradle: https://gradle.org

```
def uml2java = trafo ( name: "uml2java", of: "Acceleo" ) {
      project "UML", "./UML.ecore"

      main "Java", "IN", "./javamodel.xmi", "JavaSource"
      target "UML", "OUT", "./umlmodel", "Table"

      model = "./UML2SlicedUML.asm"
}

java2uml.dependsOn "java2javamodel"
uml2slicedml.dependsOn "java2uml"
uml2refineduml.dependsOn "uml2slicedml"
uml2java.dependsOn "uml2refineduml"
```

If a transformation is scheduled for execution, it runs through four phases: (i) setup, (ii) pre-work, (iii) work, and (iv) post-work. The first phase is dedicated to initialize a transformation by providing all the required parameters. In the above Listing, the parameters required to setup the transformation are enclosed by the curly brackets. The other three phases are part of the execution of a defined transformation in the chain. Developers can interfere in these phases by passing method calls to them, which is useful, for instance, to save intermediate results, add assertions that need to be fulfilled, provide debug information, and pass the behaviour of a transformation directly via the chain. The following Listing shows how the three phases can be accessed.

```
java2javamodel << {
    // before the transformation is executed
}

java2javamodel.work = {
    // the transformation itself
}

java2javamodel >> {
    // after the transformation has been executed
}
```

3.2 Technical Specification

The transformation composition language uses the following components and technologies:

- **eu.artist.migration.transformation.composition.ui:** provides an user interface to execute a transformation chain
- **eu.artist.migration.transformation.composition.core:** implements the transformation composition language based on Groovy, provides a translation of defined transformation chains into a Petri net representation, and enables the execution of transformation chains
- **eu.artist.migration.transformation.composition.delegates:** delegates to the supported technologies such as Acceleo, ATL, ATL/EMFTVM, Modisco
- **Groovy**: it is used as a host language for implementing the transformation compositin language
- **Java / JVM**: it is used to implement the delegators to the supported technologies; Groovy is dynamically compiled to JVM bytecode

3.3 User Manual

In this Section, we give an overview of how the transformation composition language can be used in practice. A dedicated menu entry is offered to execute defined transformation chain (right click on *.moola script and select "Run Moola")

3.3.1 Defining transformation chains

Transformation chains are composed of transformations defined in a *.moola script. The following example composes three transformations that are executed one after each other.

```
def a = trafo (  name: "a" ) {
        // setup configuration for $name"
}

def b = trafo (  name: "b" ) {
        // setup configuration for $name"
}

def c = trafo (  name: "c" ) {
        // setup configuration for $name"
}

// define the dependencies between the transformations
b.dependsOn "a"
c.dependsOn "b"
```

Dependencies between transformations can also be defined within the signature of a transformation.

```
def a = trafo (  name: "a" ) {
        // setup configuration for $name"
}

def b = trafo (  name: "b", dependsOn: ['a'] ) {
        // setup configuration for $name"
}

def c = trafo (  name: "c", dependsOn: ['b'] ) {
        // setup configuration for $name"}
```

Moreover, a short cut can be used to define the order in which the transformations shall be executed. Finally, it is possible to explicitly define the entry point of the transformation chain.

```
run inOrder: ["a", "b", "c"]
start with: ["b"]
```

3.3.2 Conditional Branching

Dependencies can also be annotated with conditions. A condition needs to forward control to exactly one succeeding transformation. For instance, if the condition of the dependency of "c" to "b" doesn't match, a default route has to be defined. An exception will be thrown if no condition matches and no default branch is defined.

```
def assertion = "some value"

def a = trafo (  name: "a" ) {
        // setup configuration for $name"
}

def a = trafo (  name: "b" ) {
        // setup configuration for $name"
        assertion = "some other value"
}

def a = trafo (  name: "c" ) {
        // setup configuration for $name"}

def a = trafo (  name: "d" ) {
        // setup configuration for $name"
}

// define the dependencies between the transformations
b.dependsOn("a")
c.dependsOn("b") {
        asseration == "some other value"
}
d.dependsOn("b")
```

3.3.3 Concurrency

Transformations in a chain are executed once all the transformations they depend on have been executed. As a result, transformations are if possible executed in parallel.

```
def a = trafo (  name: "a" ) {
        // setup configuration for $name"}
}

def b = trafo (  name: "b" ) {
        // setup configuration for $name"}
}

def c = trafo (  name: "c" ) {
        // setup configuration for $name"}
}

b.dependsOn "a"
c.dependsOn "a"
```

3.3.4 Supported Technologies

Currently, four different types of transformations are supported. They refer to the respective Eclipse-based technologies: Acceleo, ATL, ATL/EMFTVM, and MoDisco.

4 Advances in Cloud Optimization Patterns

In the ARTIST project, transformation techniques are applied in combination with annotation-based modeling [8] for realizing optimization patterns as contributed by WP9. In following, we report on a selected set of patterns for which we have developed concrete transformations and profiles, aiming to turn on-premise software into cloud-based software and optimize certain software artefacts for a selected cloud environment.

4.1 Key-Value Storage Pattern

4.1.1 Functional Description

This pattern aims to change the architecture of the database by converting a relational schema into a key-value storage. In order to successfully apply this pattern, first the domain model of the application has to be extracted, as it is described in D9.3. For the Google App Engine, we use the Objectify profile, shown in Figure 6. The transformations explained in D9.3 automatically annotate the corresponding entity classes and properties with the corresponding annotations (@Entity, @Embed, @Id…) as shown in Figure 7 for an example model of the Petstore. Finally, when the code generator is launched, the corresponding annotations are automatically created for the generated classes and properties (cf. Figure 8).

4.1.2 Technical Description

As explained in D9.3, the objectification component for generating Objectify-based PSMs from PIMs that capture the domain model of an application is realized based on two model-to-model transformations:

- **DomainModelObjectification:** Generic transformation rules to apply Objectify-based stereotypes to domain model elements.
- **DataAccessObjectsObjectification:** Generic transformation rules to generate service classes for entities of a domain model.

The main technologies used to realize the objectification component are as follows:

- **UML metamodel**: for representing the discovered UML models, which is included in the UML2 plugin
- **ATL / EMFTVM**: for implementing purposes
- **Java**: for implementation purposes.

Furthermore, and as explained in D9.5, we have extended the Acceleo-based code generator developed by Obeo Networks, which includes the following aspects:

- **Generic Code Generator**: We have added support for UML profiles to the Acceleo-based code generator developed by Obeo Networks. In this respect, we have also restructured the code templates as we produce Java annotations for corresponding UML stereotypes.
- **Code Generator Extension for Objectify**: We have implemented an Acceleo-based code generator extension for the Objectify framework.
- **Code Generator UI**: Code generator UI developed by Obeo Networks.
- **Acceleo**: Base technology to realize the code generation facility
- **Java**: Base technology to realize the code generation facility
- **UML**: Modeling language to represent the models and profiles injected to the code generation facility

4.1.3 User Manual

The steps to obtain the domain model with the Objectify profile and the corresponding stereotypes applied are explained in Section 3.3.1.3 of D9.3. Considering the generation of the code, it is explained in Section 3.2.4 of D9.5.

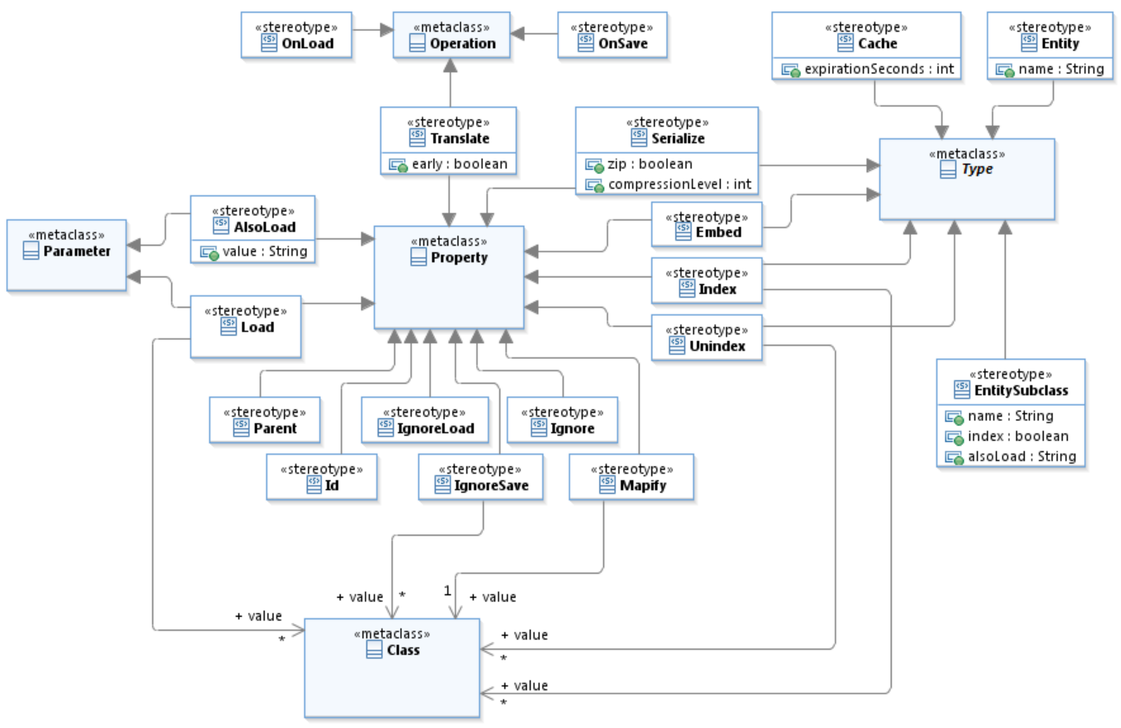
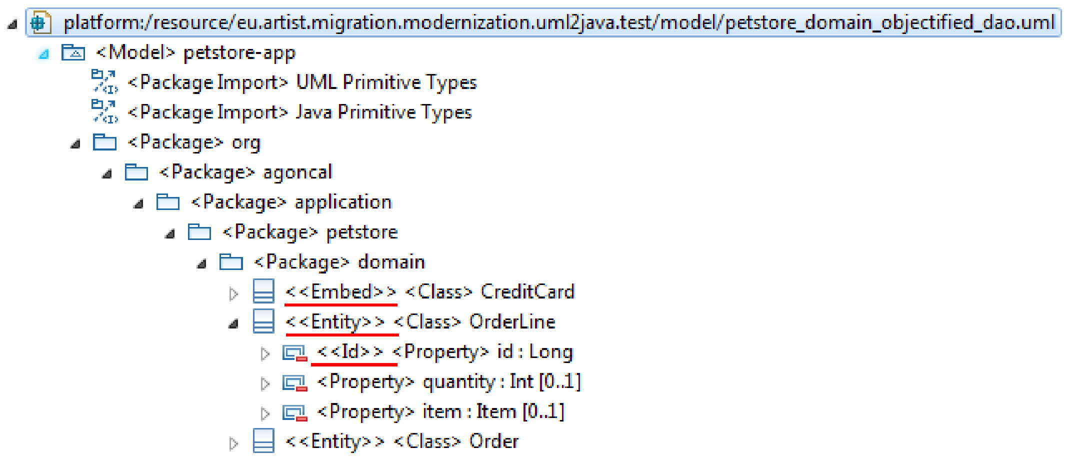
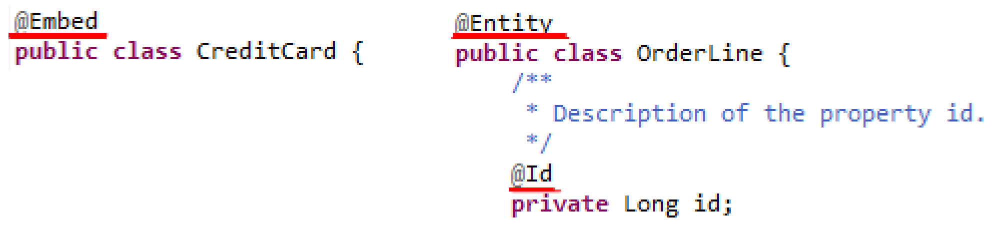


Figure 6 – UML Profile for Objectify



```
platform:/resource/eu.artist.migration.modernization.uml2java.test/model/petstore_domain_objectified_dao.uml
 <Model> petstore-app
   <Package Import> UML Primitive Types
   <Package Import> Java Primitive Types
   <Package> org
     <Package> agoncal
       <Package> application
         <Package> petstore
           <Package> domain
             <<Embed>> <Class> CreditCard
             <<Entity>> <Class> OrderLine
               <<Id>> <Property> id : Long
               <Property> quantity : Int [0..1]
               <Property> item : Item [0..1]
             <<Entity>> <Class> Order
```

Figure 7 Model with the corresponding annotations for the *Key-Value Storage* pattern



```
@Embed
public class CreditCard {
```

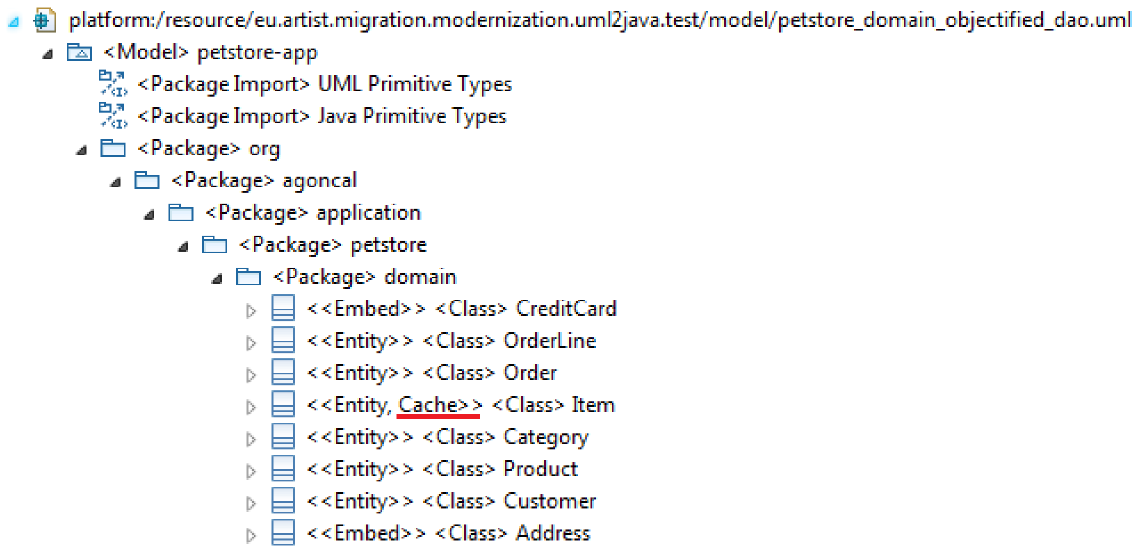
```
@Entity
public class OrderLine {
    /**
     * Description of the property id.
     */
    @Id
    private Long id;
```

Figure 8 –Annotations automatically generated for the *Key-Value Storage* pattern in GAE

4.2 Caching Pattern

4.2.1 Functional Description

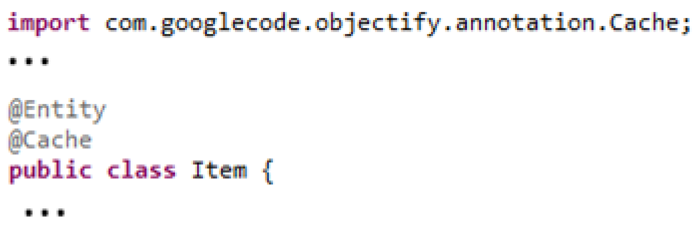
The application of this pattern aims at reducing the time for performing reads to the database, facilitating database scalability. In the Google App Engine, we simply need the *<<Cache>>* stereotype of the Objectify profile shown in Figure 6 on the entities that we want to be cached. Thus, we only need to know which classes, out of those representing the domain model, we want to be cached, and stereotype them, as shown in for the *Item* class in Figure 9.



```
platform:/resource/eu.artist.migration.modernization.uml2java.test/model/petstore_domain_objectified_dao.uml
  <Model> petstore-app
    <Package Import> UML Primitive Types
    <Package Import> Java Primitive Types
    <Package> org
      <Package> agoncal
        <Package> application
          <Package> petstore
            <Package> domain
              <<Embed>> <Class> CreditCard
              <<Entity>> <Class> OrderLine
              <<Entity>> <Class> Order
              <<Entity, Cache>> <Class> Item
              <<Entity>> <Class> Category
              <<Entity>> <Class> Product
              <<Entity>> <Class> Customer
              <<Embed>> <Class> Address
```

Figure 9 Class annotated with the *Cache* stereotype

Then, the corresponding @Cache annotation at code level as well as the necessary import is automatically generated by the code generator (cf. Figure 10).



```
import com.googlecode.objectify.annotation.Cache;
...
@Entity
@Cache
public class Item {
 ...
```

Figure 10 Annotation automatically generated for the *Cache* pattern in GAE

4.2.2 Technical Description

As explained in D9.5 [1], we have extended the Acceleo-based code generator developed by Obeo Networks. For this specific pattern, we add support for including the *@Cache* annotation and the corresponding imports.

4.2.3 User Manual

In order to execute this pattern for having the annotations included in the code, the user needs to execute the extended code generator after the <<Cache>> stereotype has been applied to the corresponding classes.

4.3 Multitenancy Pattern

4.3.1 Functional Description

As we explained in D9.3, **t**he Google App Engine offers a PaaS service to its developers, and we do not have to worry about splitting the data in the database among tenants, since GAE manages it automatically and transparently to the developer. GAE offers the *Namespaces* service[4] to implement multitenancy. By using this service, namespaces can be set, obtained, and validated using *namespace_manager*.

Although in D9.3 we presented the possibility to stereotype both operations and classes with the *Multitenancy* profile, we only consider the stereotype of classes in order to automatically generate code that supports multitenancy. The profile that we propose is shown in Figure 11. When a *Class* is stereotyped with *Multitenancy*, this means that all the entities of this specific type consider multitenancy. Consequently, all the reads and writes on these entities will be done by applying the ID of the specific tenant or group of tenants. At this level, it can also be specified the type and level of multitenancy that the system must provide, although for GAE, these parameters are not needed.

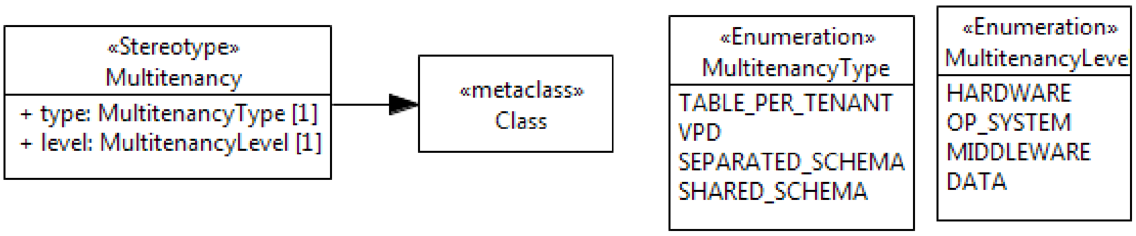
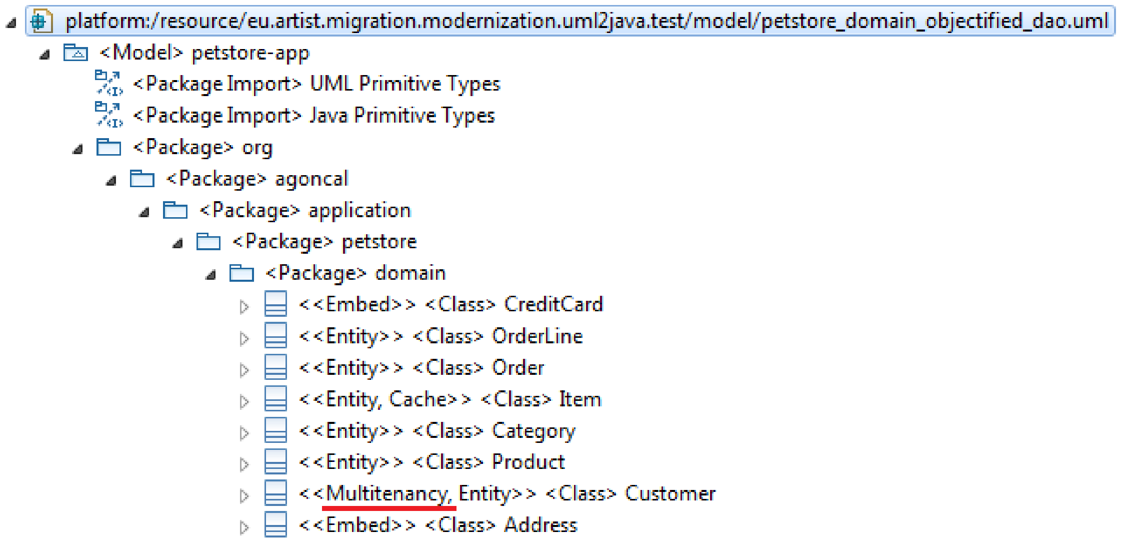


Figure 11 Steoreotype for *Multitenancy*

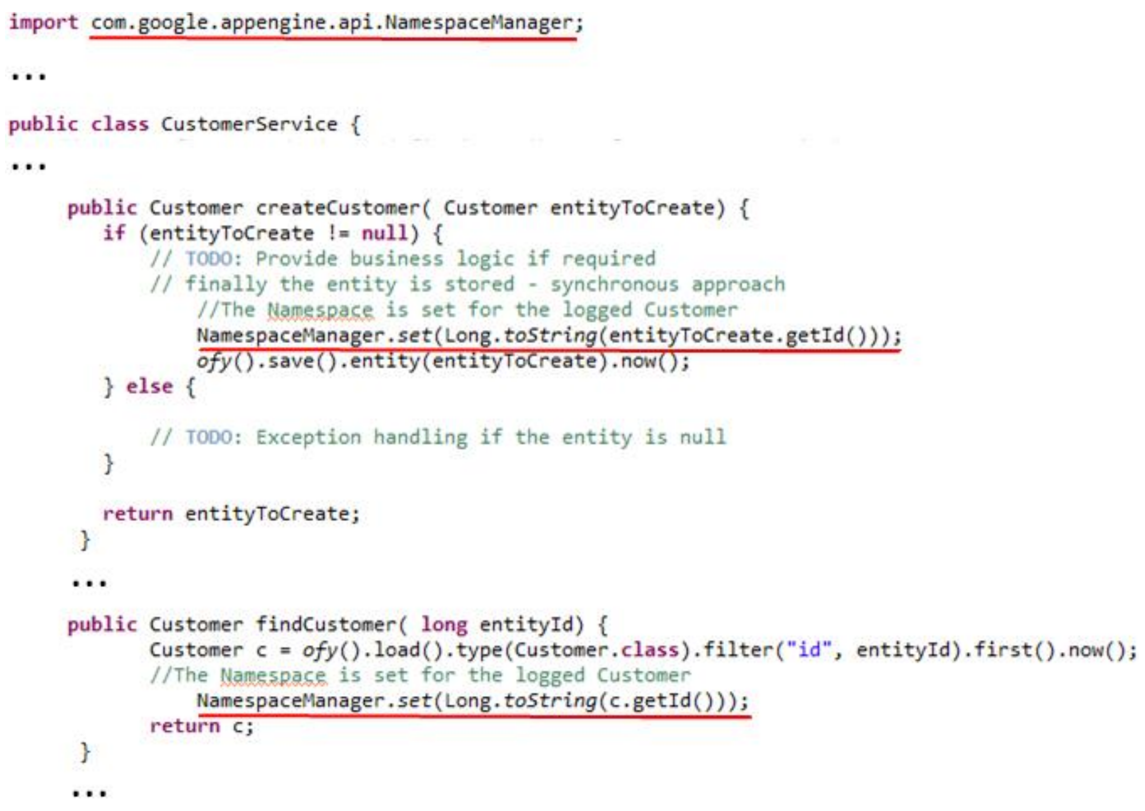
In order to automatically generate code that supports multitenancy, we have extended the Acceleo code generator to include some code related to multitenancy in some service classes of those entities stereotyped with *<<Multitenancy>>*. The user should be aware that the entities that should be stereotyped with *<<Multitenancy>>* are those that must have their data split in the database. In GAE, the *Namespace* is set to be the ID of the user who is logged in the application. For instance, in the model of the Petstore shown in Figure 12, we can see how the *Customer* class is extended with the *Multitenancy* stereotype, since the user wants each customer of the application to read/write his/her own data. This means that, when we execute the code generator, some code specific for the *NamespaceManager* is added in the service class of the Customer. In fact, we consider that a new customer logs in whenever it is created or it is searched for. For this reason, these two operations of the *CustomerService* class get extended. The necessary imports to support this service are also included automatically (cf. Figure 13).

[4] https://developers.google.com/appengine/docs/python/multitenancy/multitenancy



```
platform:/resource/eu.artist.migration.modernization.uml2java.test/model/petstore_domain_objectified_dao.uml
  <Model> petstore-app
      <Package Import> UML Primitive Types
      <Package Import> Java Primitive Types
    <Package> org
      <Package> agoncal
        <Package> application
          <Package> petstore
            <Package> domain
              <<Embed>> <Class> CreditCard
              <<Entity>> <Class> OrderLine
              <<Entity>> <Class> Order
              <<Entity, Cache>> <Class> Item
              <<Entity>> <Class> Category
              <<Entity>> <Class> Product
              <<Multitenancy, Entity>> <Class> Customer
              <<Embed>> <Class> Address
```

Figure 12 ***Customer*** **class annotated with the** ***Multitenancy*** **stereotype**



```
import com.google.appengine.api.NamespaceManager;

...

public class CustomerService {

...

    public Customer createCustomer( Customer entityToCreate) {
        if (entityToCreate != null) {
            // TODO: Provide business logic if required
            // finally the entity is stored - synchronous approach
                //The Namespace is set for the logged Customer
                NamespaceManager.set(Long.toString(entityToCreate.getId()));
                ofy().save().entity(entityToCreate).now();
        } else {

            // TODO: Exception handling if the entity is null
        }

        return entityToCreate;
    }

    ...

    public Customer findCustomer( long entityId) {
            Customer c = ofy().load().type(Customer.class).filter("id", entityId).first().now();
            //The Namespace is set for the logged Customer
                NamespaceManager.set(Long.toString(c.getId()));
            return c;
    }

    ...
```

Figure 13 Some code generated for the ***Multitenancy*** **pattern in the** ***CustomerService*** **class**

4.3.2 Technical Description

As explained in D9.5 [1], we have extended the Acceleo-based code generator developed by Obeo Networks. For this specific pattern, we add support for extending specific service classes and for adding the necessary imports.

4.3.3 User Manual

In order to execute this pattern for adding the necessary code, the user needs to execute the extended code generator after the <<Multitenancy>> stereotype has been applied to the corresponding domain classes.

4.4 Materialized View Pattern

4.4.1 Functional Description

When a query requires only a subset of the data from some entities, such as a summary of orders for several customers without all of the order details, it must extract all of the data for the relevant entities in order to extract the required information. In addition to joining tables or combining data entities, queries may also include the current values of calculated columns or data items. Realizing the same query once and again influences negatively the performance of the system. This pattern proposes to store the results of these frequently-realized queries in the Cache, improving the performance of the application.

We consider that these queries are realized inside specific methods of the classes in the migrated application. Consequently, we stereotype with *<<MaterializedView>>* those methods that contain queries of this type (cf. Figure 14). The idea is to first look if the value we are interested in is in the Cache. If it is not, the query is realized and the value returned (and also stored in the Cache). If it is, it is retrieved from the Cache.

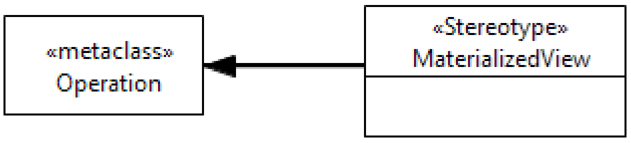
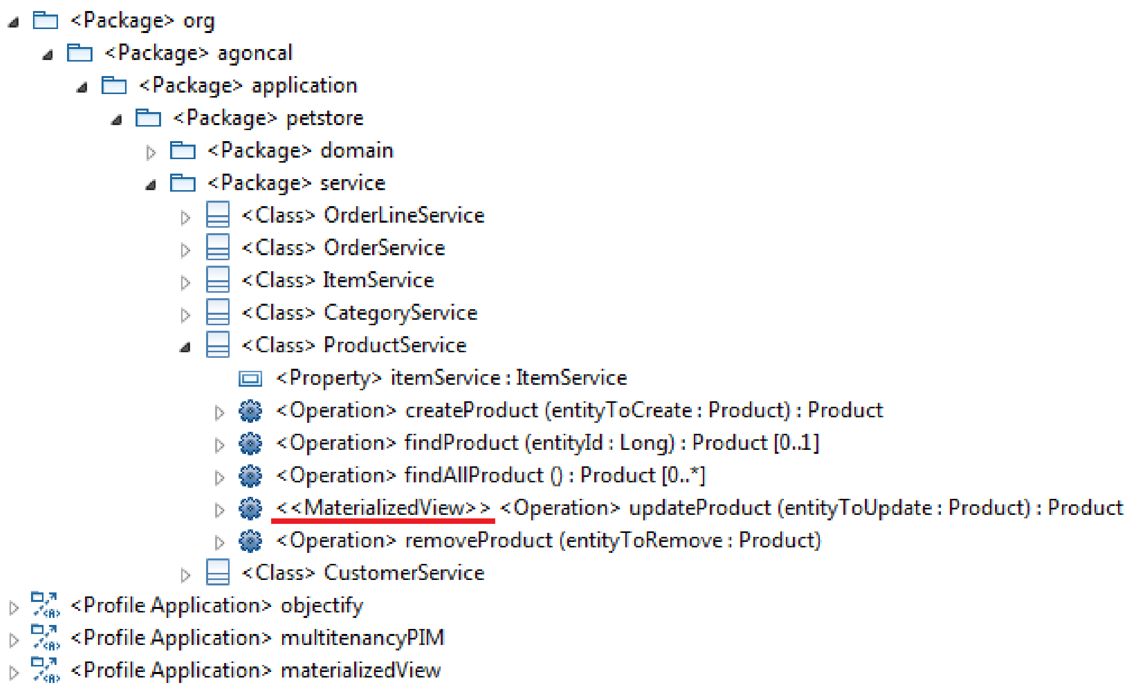


Figure 14 Stereotype for MaterializedView pattern

For automatically generating code for this pattern, in order for an operation to acquire the logic to implement the materialized view pattern, it needs to have the *<<MaterializedView>>* stereotype applied, as shown for the *updateProduct* operation in Figure 15. Then, the corresponding code template in code level is automatically generated. Then, when executing the code generator, the necessary imports as well as the code are automatically generated, as shown in Figure 16.



```
<Package> org
  <Package> agoncal
    <Package> application
      <Package> petstore
        <Package> domain
        <Package> service
          <Class> OrderLineService
          <Class> OrderService
          <Class> ItemService
          <Class> CategoryService
          <Class> ProductService
            <Property> itemService : ItemService
            <Operation> createProduct (entityToCreate : Product) : Product
            <Operation> findProduct (entityId : Long) : Product [0..1]
            <Operation> findAllProduct () : Product [0..*]
            <<MaterializedView>> <Operation> updateProduct (entityToUpdate : Product) : Product
            <Operation> removeProduct (entityToRemove : Product)
          <Class> CustomerService
<Profile Application> objectify
<Profile Application> multitenancyPIM
<Profile Application> materializedView
```

Figure 15 Operation annotated with the *MaterializedView* stereotype

```
import com.google.appengine.api.memcache.MemcacheService;
import com.google.appengine.api.memcache.MemcacheServiceFactory;
import com.google.appengine.api.memcache.ErrorHandlers;

...
public Product updateProduct( Product entityToUpdate) {

    /**************The code for the Materialized View Pattern begins here*************/
    String key = "MV1"; //This key needs to be unique
    MemcacheService syncCache = MemcacheServiceFactory.getMemcacheService();
    syncCache.setErrorHandler(ErrorHandlers.getConsistentLogAndContinue(Level.INFO));
    String value = (String) syncCache.get(key);
    if (value == null) {
        // TODO: Provide the operation whose result is then materialized in a view in the cache
        syncCache.put(key, value);
    }else{
        // The value was already in the cache, we have therefore retrieved it, now something can be done with it
    }
    /**************The code for the Materialized View Pattern ends here*************/

    ...
```

Figure 16 Code automatically generated for the *MaterializedView* pattern

4.4.2 Technical Description

As explained in D9.5 [1], we have extended the Acceleo-based code generator developed by Obeo Networks. For this specific pattern, we add support for extending the specific operations of service classes that are annotated with <<MaterializedView>>. Specifically, the extension in code level refers to the inclusion of the template in said operations.

4.4.3 User Manual

In order to execute this pattern for having the code included in the operations, the user needs to execute the extended code generator after the <<MaterializedView>> stereotype has been applied to the corresponding operations.

4.5 Circuit Breaker Pattern

4.5.1 Functional Description

A circuit breaker acts as a proxy for operations that may fail. The proxy should monitor the number of recent failures that have occurred, and then use this information to decide whether to allow the operation to proceed or simply return an exception immediately or wait for a specific timeout for retrying the operation.

We have implemented this pattern in a model level as a stereotype extending an operation, since it is within methods where the calls to a specific service may fail. As we need the information about the number of failures before the circuit breaker acts, we have set an attribute *threshold* in the stereotype. We also need to know the time we want the application to wait before retrying the operation, something that is stored in the *timeout* attribute. The stereotype we have defined is shown in Figure 17.

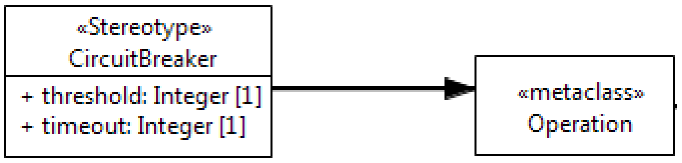
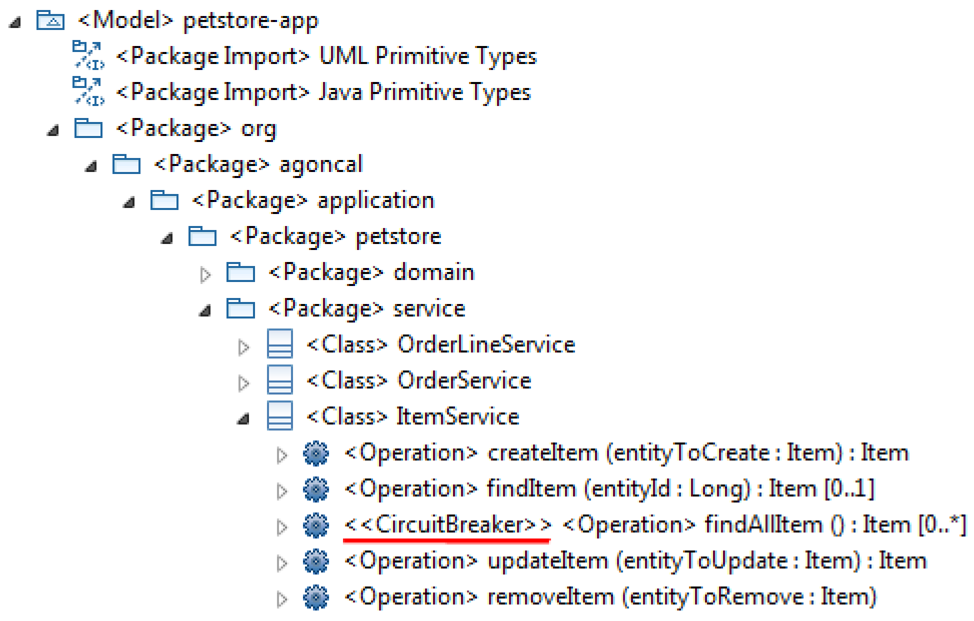


Figure 17 Stereotype for the *CircuitBreaker* pattern

In order for an operation to acquire the logic to implement the circuit breaker pattern, it needs to have the *<<CircuitBreaker>>* stereotype applied (cf. Figure 18). Then, the corresponding code template in code level is automatically generated together with the *CircuitBreaker.java* and *ServiceCircuit.java* classes.



```
<Model> petstore-app
    <Package Import> UML Primitive Types
    <Package Import> Java Primitive Types
    <Package> org
        <Package> agoncal
            <Package> application
                <Package> petstore
                    <Package> domain
                    <Package> service
                        <Class> OrderLineService
                        <Class> OrderService
                        <Class> ItemService
                            <Operation> createItem (entityToCreate : Item) : Item
                            <Operation> findItem (entityId : Long) : Item [0..1]
                            <<CircuitBreaker>> <Operation> findAllItem () : Item [0..*]
                            <Operation> updateItem (entityToUpdate : Item) : Item
                            <Operation> removeItem (entityToRemove : Item)
```

Figure 18 Operation annotated with the *CircuitBreaker* stereotype

4.5.2 Technical Description

As explained in D9.5 [1], we have extended the Acceleo-based code generator developed by Obeo Networks. For this specific pattern, we add support for extending the specific operations of service classes that are annotated with <<CircuitBreaker>>. Specifically, the extension at code level refers to the inclusion of the template in said operations.

4.5.3 User Manual

In order to execute this pattern for having the code included in the operations, the user needs to execute the extended code generator after the <<CircuitBreaker>> stereotype has been applied to the corresponding operations.

4.6 Cloudification of Resource Monitoring Concerns

4.6.1 Functional Description

Platform resource monitoring in Java is possible through several means, the most widely used being JMX (Java Management eXtension). JMX provides means to add flexible and powerful management interfaces to an application. Through JMX technology, a set of platform resources to be controlled can be seen as simple, well-defined objects whose properties map to the lower-level characteristics of the given resource. With JMX, Java instruments the Virtual Machine itself to provide visibility into the state of memory management, class loading, active threads, logging, and platform configuration. This functionality is provided through *MXBeans*, which are, in effect, management beans encapsulating specific parts of the Java Platform's internals. The management interface exposes several classes to control a wide set of resources, including Memory, Operating System, Threads, Runtime.

When moving to cloud, in particular to PaaS providers, it is possible that this information is no longer available, incorrect or not supported. The transformation described in this section is aimed at replacing the existing technology to gather the information about platform resources with one that is supported by the cloud provider, with a minimal impact for the rest of the system.

Although for some types of resources resource management is taken care of by the cloud provider, the monitoring of resource consumption can be useful to systems when they want to enact closed loop management mechanisms, or even to try to contain costs related to resources consumption.

Taking as an example Google App Engine (GAE), while this cloud provider does not support JMX, including *JMXBeans* access, it offers billing schemas based on resource usage quotas. Resource monitoring can be useful to check that the system does not exceed the thresholds over which resources are to be paid for. Even if JMX is not supported as such, GAE offers several other services to manage and monitor the consumption of resources. The idea of the transformation is to create a profile that allows identifying platform resources, so to be able to replace the technology to access them with the ones compatible with the cloud provider.

In Figure 19, the whole transformation chain is visualized, together with the artifacts involved in the process. All artifacts and transformations are described in more detail in the following paragraphs. The initial input to the transformation process is the UML Model of the system to be migrated. The model is annotated with the *ResourceMonitoring* Profile created to identify resources ant its properties. This annotated model, still platform independent, is then mapped with the Resource Monitoring GAE library model. This library is specific to Google app engine and uses cloud provider specific APIs to gather information on the resources.

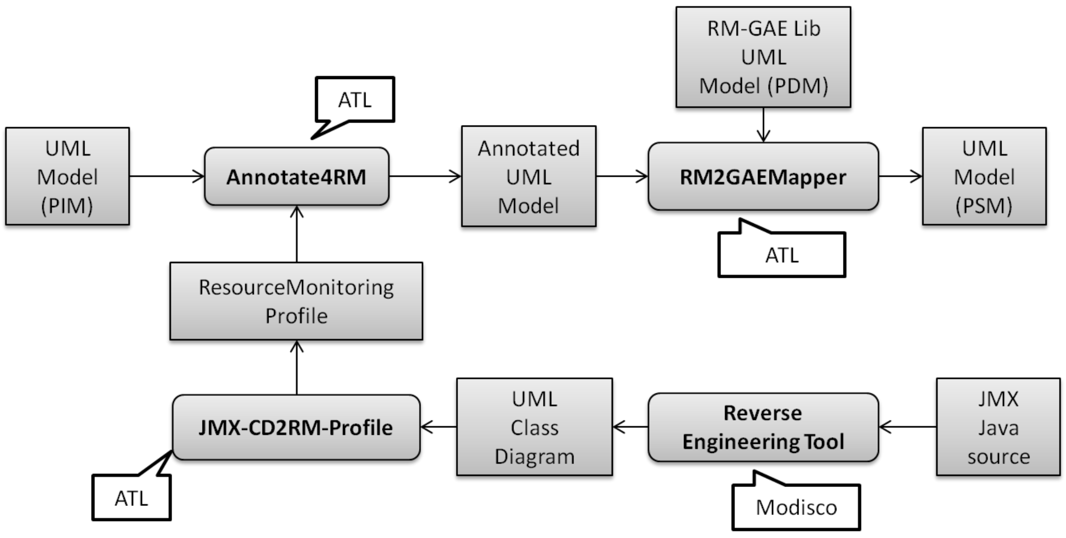


Figure 19 Transformation Chain

The result of this mapping is a Platform specific model with resource management support for the target cloud provider, in this case Google App Engine

4.6.1.1 Platform-Independent Profile for Resource Monitoring

The Resource Monitoring profile (*res-model.profile*) has been derived automatically from the JMX specifications. It describes resources and their main properties relevant at the level of the Java platform. The profile has been generated using an ATL transformation (*JMX-CD2RM-Profile)* that maps the JMX concepts to stereotypes, along with their properties. The diagram in Figure 20 depicts some of the resources modelled in the profile. Even if not all providers support the access to all the properties listed in the *res-model.profile*, the profile is as complete as possible to be platform independent.

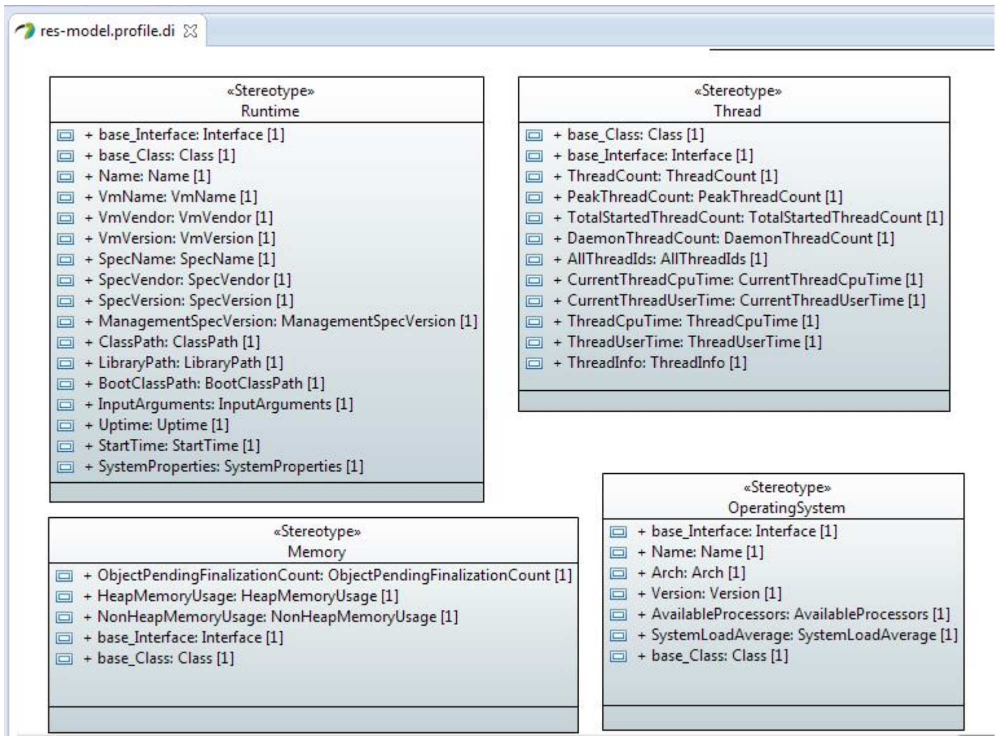

res-model.profile.di
«Stereotype»
Runtime
+ base_Interface: Interface [1]
+ base_Class: Class [1]
+ Name: Name [1]
+ VmName: VmName [1]
+ VmVendor: VmVendor [1]
+ VmVersion: VmVersion [1]
+ SpecName: SpecName [1]
+ SpecVendor: SpecVendor [1]
+ SpecVersion: SpecVersion [1]
+ ManagementSpecVersion: ManagementSpecVersion [1]
+ ClassPath: ClassPath [1]
+ LibraryPath: LibraryPath [1]
+ BootClassPath: BootClassPath [1]
+ InputArguments: InputArguments [1]
+ Uptime: Uptime [1]
+ StartTime: StartTime [1]
+ SystemProperties: SystemProperties [1]
«Stereotype»
Thread
+ base_Class: Class [1]
+ base_Interface: Interface [1]
+ ThreadCount: ThreadCount [1]
+ PeakThreadCount: PeakThreadCount [1]
+ TotalStartedThreadCount: TotalStartedThreadCount [1]
+ DaemonThreadCount: DaemonThreadCount [1]
+ AllThreadIds: AllThreadIds [1]
+ CurrentThreadCpuTime: CurrentThreadCpuTime [1]
+ CurrentThreadUserTime: CurrentThreadUserTime [1]
+ ThreadCpuTime: ThreadCpuTime [1]
+ ThreadUserTime: ThreadUserTime [1]
+ ThreadInfo: ThreadInfo [1]
«Stereotype»
Memory
+ ObjectPendingFinalizationCount: ObjectPendingFinalizationCount [1]
+ HeapMemoryUsage: HeapMemoryUsage [1]
+ NonHeapMemoryUsage: NonHeapMemoryUsage [1]
+ base_Interface: Interface [1]
+ base_Class: Class [1]
«Stereotype»
OperatingSystem
+ base_Interface: Interface [1]
+ Name: Name [1]
+ Arch: Arch [1]
+ Version: Version [1]
+ AvailableProcessors: AvailableProcessors [1]
+ SystemLoadAverage: SystemLoadAverage [1]
+ base_Class: Class [1]

Figure 20 Platform Independent Profile for Resource Monitoring

4.6.1.2 Platform-Specific model for Resource Monitoring

To exploit the information made available from Google App Engine, a library for resource monitoring has been developed in the context of ARTIST Project. This library uses platform specific services (GAE QuotasService, Module APIs, GAE SystemProperty class) plus other standard Java calls, to gather information about resources.

A model for this library has been generated and annotated with the Resource profile, and used as the PDM for this transformation

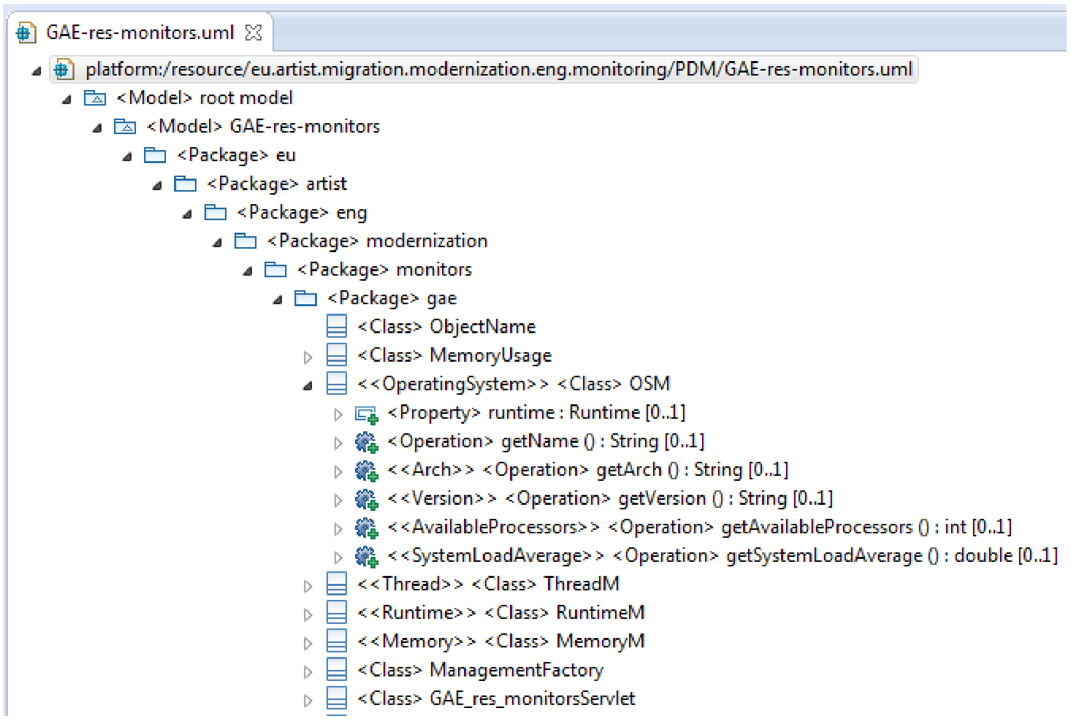

```
GAE-res-monitors.uml
platform:/resource/eu.artist.migration.modernization.eng.monitoring/PDM/GAE-res-monitors.uml
  <Model> root model
    <Model> GAE-res-monitors
      <Package> eu
        <Package> artist
          <Package> eng
            <Package> modernization
              <Package> monitors
                <Package> gae
                  <Class> ObjectName
                  <Class> MemoryUsage
                  <<OperatingSystem>> <Class> OSM
                    <Property> runtime : Runtime [0..1]
                    <Operation> getName () : String [0..1]
                    <<Arch>> <Operation> getArch () : String [0..1]
                    <<Version>> <Operation> getVersion () : String [0..1]
                    <<AvailableProcessors>> <Operation> getAvailableProcessors () : int [0..1]
                    <<SystemLoadAverage>> <Operation> getSystemLoadAverage () : double [0..1]
                  <<Thread>> <Class> ThreadM
                  <<Runtime>> <Class> RuntimeM
                  <<Memory>> <Class> MemoryM
                  <Class> ManagementFactory
                  <Class> GAE_res_monitorsServlet
```

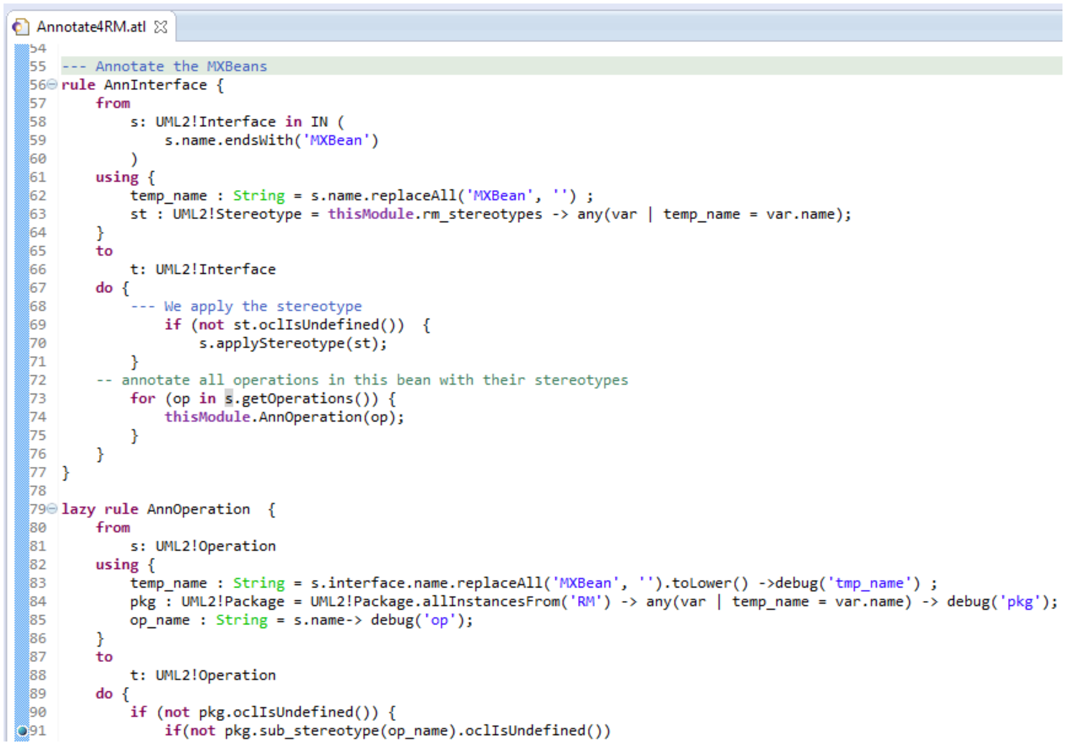
Figure 21 GAE resource monitors PDM

4.6.2 Technical Specifications

This section describes an example of migration of a SPCoop service responsible for monitoring and managing resources using JMX platform beans, to the corresponding GAE compliant service.

The input Platform Independent class diagram represents the system once sliced to extract the relevant view of the original PSM. This is the output of the model discovery / understanding phase of the ARTIST methodology, and input to the transformation described in this section.

Taking into consideration the migration goal described above, the first step of the migration is to map resources in JMX and map them to the vocabulary of the Resource monitor PIM profile. For this purpose an in-place ATL transformation (*Annotate4RM.atl*) has been created. A snippet of the transformation, regarding the mapping of Interfaces is shown below:



Annotate4RM.atl

```
--- Annotate the MXBeans
rule AnnInterface {
    from
        s: UML2!Interface in IN (
            s.name.endsWith('MXBean')
        )
    using {
        temp_name : String = s.name.replaceAll('MXBean', '') ;
        st : UML2!Stereotype = thisModule.rm_stereotypes -> any(var | temp_name = var.name);
    }
    to
        t: UML2!Interface
    do {
        --- We apply the stereotype
            if (not st.oclIsUndefined())  {
                s.applyStereotype(st);
        }
    -- annotate all operations in this bean with their stereotypes
        for (op in s.getOperations()) {
            thisModule.AnnOperation(op);
        }
    }
}

lazy rule AnnOperation  {
    from
        s: UML2!Operation
    using {
        temp_name : String = s.interface.name.replaceAll('MXBean', '').toLower() ->debug('tmp_name') ;
        pkg : UML2!Package = UML2!Package.allInstancesFrom('RM') -> any(var | temp_name = var.name) -> debug('pkg');
        op_name : String = s.name-> debug('op');
    }
    to
        t: UML2!Operation
    do {
        if (not pkg.oclIsUndefined()) {
            if(not pkg.sub_stereotype(op_name).oclIsUndefined())
```

Figure 22 Annotate4RM transformation

A sample of the resulting PIM model with RM stereotypes is shown in Figure 23.

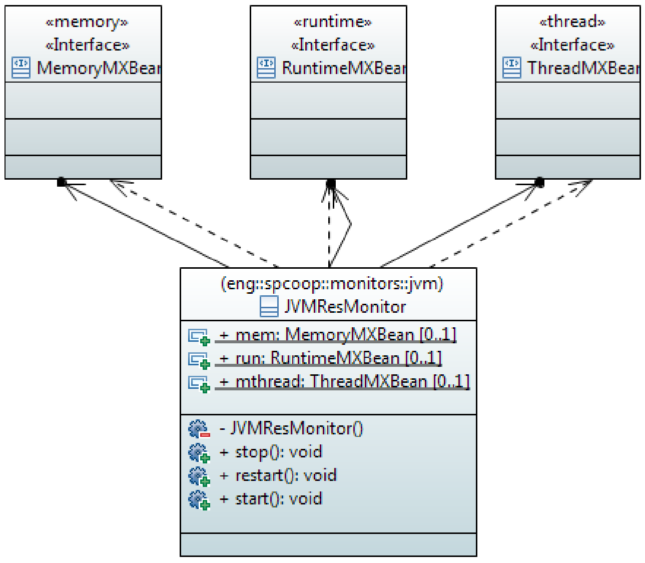


Figure 23 PIM model annotated with Resource Monitor profile

The following step consists on merging the annotated PIM model with a Platform Dependent Model that links concepts of the PIM profile to the classes needed to resource properties values in Google App Engine as already shown in Figure 2124.

The transformation is performed by applying the respective transformation – `RM2GAEMapper.atl` – that relates classes, attributes and operations of the PIM profile with correspondent ones of the PDM. The resulting Platform Specific diagram is depicted in Figure 25.

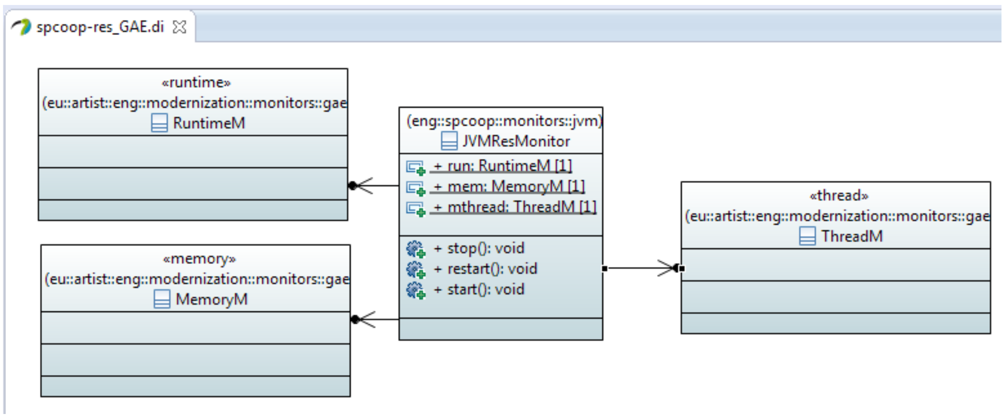


Figure 24 SPCoop monitoring service PSM (GAE compliant)

4.6.3 User Manual

The concrete implementations of M2MT modules supporting the "cloudification" of resource monitoring in Google App Engine are part of the ARTIST Cloudification Toolbox, and located in ARTIST-Tooling Github Repository, under the folder *migration/modernization/* in the project *eu.artist.migration.modernization.eng.monitor.*

The project contains the following folders:

- *transform:* containing all ATL transformations;
- *models:* collects input and output models. The input PIM is *spcoop-res_JMX.uml,* while the PSM model produced by the transformation is named *spcoop-res_GAE.uml*;
- *PDM:* contains the resource-monitor-GAE.uml model;
- *profiles:* contains the *res-model.profile*
- *launch:* The folder contains the launch files allowing to run the transformations. Files are numbered for convenience of the user.

In order to run the transformations, launch file *1_RM_annotate.launch* first, to create the *spcoop-res_PIM_RM.uml* intermediate model, then launch *2_PIM2PSM.launch* to generate the target PSM model. To launch the file run right click on the file in the workspace resource tree and select "Run as", choosing the launch configuration matching the file name.

4.7 Federated Identity Pattern

4.7.1 Functional Description

Elaborating the work described in deliverable D9.3 on the cloudification of security concerns, we have extended the transformation to include the model-to-code step. Figure 25 recalls the transformation chain already covered in D9.3 (in grey) and highlights the extension, the implementation on top of an Acceleo-based code generator for the Federated Identity pattern.

4.7.2 Technical Description

This generator extends the standard UML2Java Acceleo code generator to create stubs of core methods, so to implement correctly the authentication process using the openID4Java library, based on the stereotypes in the Authentication Enforcer pattern.

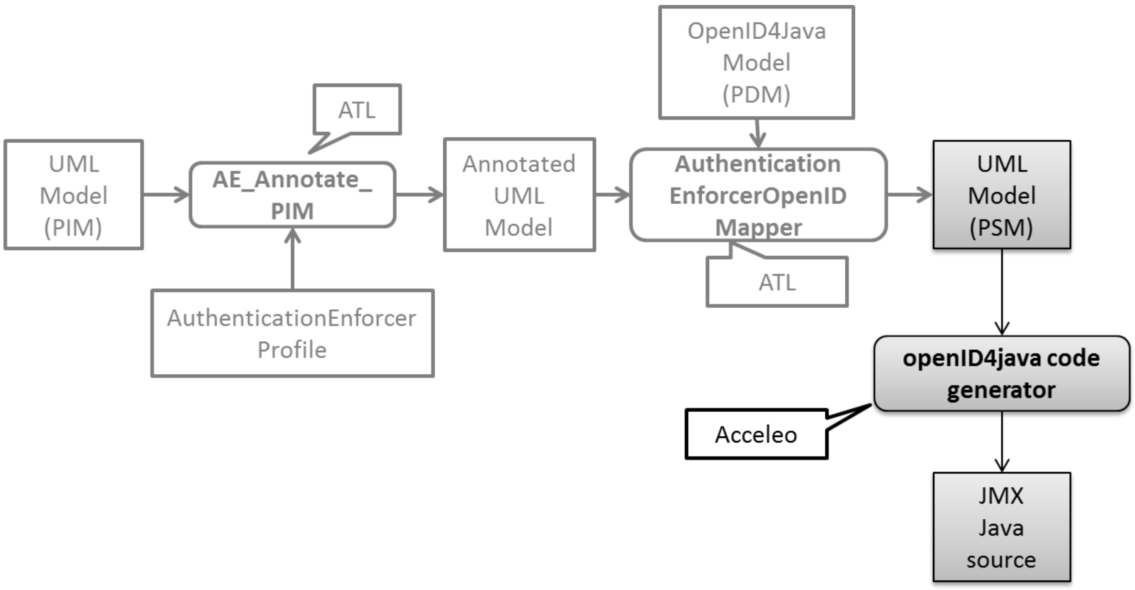
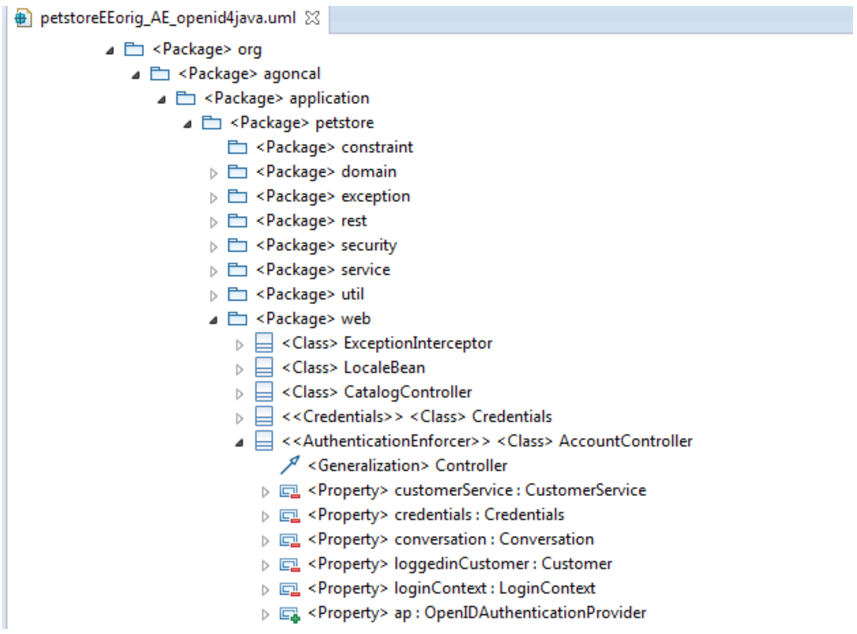


Figure 25 Federated Identity optimization pattern process

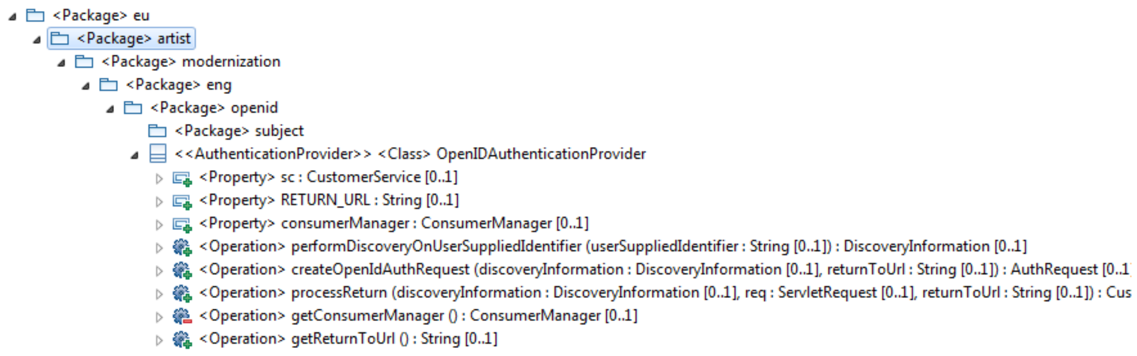
In particular, the core operations have method bodies' templates and comments that guide the correct interaction with the openid4java library.



```
petstoreEEorig_AE_openid4java.uml
  <Package> org
    <Package> agoncal
      <Package> application
        <Package> petstore
          <Package> constraint
          <Package> domain
          <Package> exception
          <Package> rest
          <Package> security
          <Package> service
          <Package> util
          <Package> web
            <Class> ExceptionInterceptor
            <Class> LocaleBean
            <Class> CatalogController
            <<Credentials>> <Class> Credentials
            <<AuthenticationEnforcer>> <Class> AccountController
              <Generalization> Controller
              <Property> customerService : CustomerService
              <Property> credentials : Credentials
              <Property> conversation : Conversation
              <Property> loggedinCustomer : Customer
              <Property> loginContext : LoginContext
              <Property> ap : OpenIDAuthenticationProvider
```

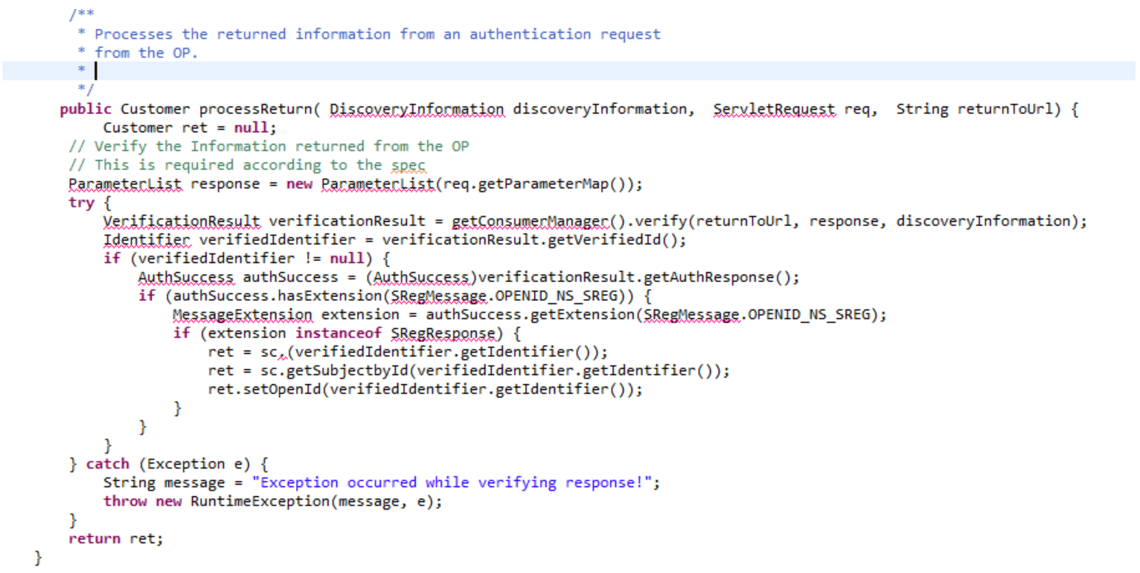
Figure 26 Petstore PSM with OpenID UML

For example, using stereotypes from the AuthenthicationEnforcer profile, it has been possible to identify and initialize methods for the *AuthenticationProvider* class, where variables referring to the *Subject* class have been matched to the correct signature. In the following an example of a model and one of its generated methods:



```
◢ <Package> eu
    ◢ <Package> artist
        ◢ <Package> modernization
            ◢ <Package> eng
                ◢ <Package> openid
                      <Package> subject
                    ◢ <<AuthenticationProvider>> <Class> OpenIDAuthenticationProvider
                        ▷ <Property> sc : CustomerService [0..1]
                        ▷ <Property> RETURN_URL : String [0..1]
                        ▷ <Property> consumerManager : ConsumerManager [0..1]
                        ▷ <Operation> performDiscoveryOnUserSuppliedIdentifier (userSuppliedIdentifier : String [0..1]) : DiscoveryInformation [0..1]
                        ▷ <Operation> createOpenIdAuthRequest (discoveryInformation : DiscoveryInformation [0..1], returnToUrl : String [0..1]) : AuthRequest [0..1
                        ▷ <Operation> processReturn (discoveryInformation : DiscoveryInformation [0..1], req : ServletRequest [0..1], returnToUrl : String [0..1]) : Cus
                        ▷ <Operation> getConsumerManager () : ConsumerManager [0..1]
                        ▷ <Operation> getReturnToUrl () : String [0..1]
```

Figure 27 Stereotyped example model



```
    /**
     * Processes the returned information from an authentication request
     * from the OP.
     *
     */
   public Customer processReturn( DiscoveryInformation discoveryInformation,  ServletRequest req,  String returnToUrl) {
        Customer ret = null;
    // Verify the Information returned from the OP
    // This is required according to the spec
    ParameterList response = new ParameterList(req.getParameterMap());
    try {
        VerificationResult verificationResult = getConsumerManager().verify(returnToUrl, response, discoveryInformation);
        Identifier verifiedIdentifier = verificationResult.getVerifiedId();
        if (verifiedIdentifier != null) {
            AuthSuccess authSuccess = (AuthSuccess)verificationResult.getAuthResponse();
            if (authSuccess.hasExtension(SRegMessage.OPENID_NS_SREG)) {
                MessageExtension extension = authSuccess.getExtension(SRegMessage.OPENID_NS_SREG);
                if (extension instanceof SRegResponse) {
                    ret = sc.(verifiedIdentifier.getIdentifier());
                    ret = sc.getSubjectbyId(verifiedIdentifier.getIdentifier());
                    ret.setOpenId(verifiedIdentifier.getIdentifier());
                }
            }
        }
    } catch (Exception e) {
        String message = "Exception occurred while verifying response!";
        throw new RuntimeException(message, e);
    }
    return ret;
}
```

Figure 28 Generated code from stereotyped example model

5 Advances in Measuring the Quality of Generated Code

5.1 Functional Description

In the first stages of the ARTIST methodology (pre-migration phase) and project, where the feasibility for a migration to cloud is being evaluated, the maintainability metric (as defined by IEEE [5]) for analysing the software complexity is calculated. In the context of the modernization phase, the maintainability metric of the generated metric is calculated too. The main purpose for this re-calculation of the maintainability index is to analyze if within the application modernization process (through model to model and model to code transformations) this metric has improved.

Maintainability is defined by IEEE standard glossary of Software Engineering [2] as “the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment”.

As explained in D5.1.1, in the context of ARTIST we use the compound MEMOOD method [6], to calculate the Maintenance based on the following model:

$$Maintenance = 2.399 + 0.493 \times Modifiability + 0.474 \times Understability + 0.524 \times Scalability + 0.507 * LOC$$

Where:

$$Modifiability = 0.629 + 0.471 \times NC - 0.173 \times NGen - 0.616 \times NAggH - 0.696 \times NGenH + 0.396 \times MaxDIT$$

$$Understability = 1.66 + 0.256 \times NC - 0.394 \times NGenH$$

$$Scalability = 0.182 \times 0.99 \times AC + 0.100 \times EC + 0.097 \times ND - 0.036 \times PC + 0.068 \times DMS$$

$$LOC = 0.269 + 0.008 \times Coupling + 0.181 \times cohesion + 0.119 \times CC + 0.084 \times ILCC$$

Being NC: Number of classes; NGen: Number of Generalizations; NGenH: Number Of Generalizations Hierarchies; AC: Afferent Coupling; EC: Efferent Coupling; ND: Nesting Depth; DMS: Distance from main sequence; PC: % coverage; CC: Cyclomatic complexity; ILCC: IL Cyclometic complexity.

This prototype calculates the final Maintainability metric through the combination of the atomic metrics. The final objective is to calculate both metrics, the one obtained from the source code and the one calculated from the generated code, and analyse possible variations.

5.2 Technical description

5.2.1 Prototype architecture

The current Maintainability metric calculator prototype architecture is a java API that explores source files to generate several Metrics of a specific project. The following image depicts the overall architecture:

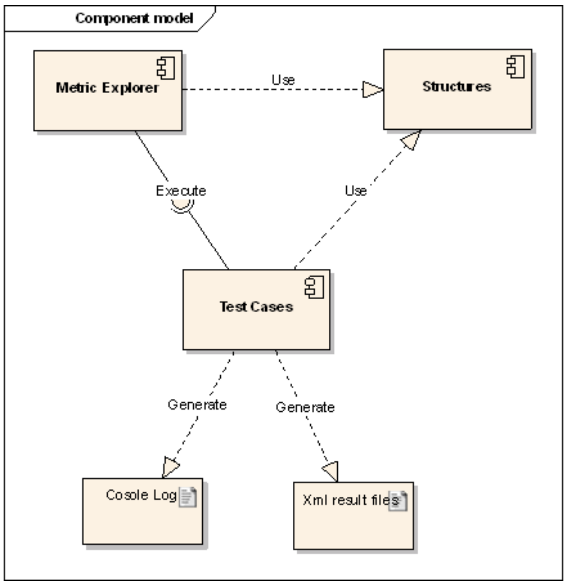


Figure 29 MMC High level architecture

While the main objective of the Maintainability Metric Calculator is to expose an API that any other plug-in or RCP could use to obtain the metrics generated in the ARTIST project, it also provides Test Cases to access the same functionality as if used programmatically. The generated metrics is available in XML files and console log.

5.2.2 Components description

The current Maintainability Metric calculator prototype component comprises three components:

- **Metric Explorer:** This is the main component of the Maintainability Metric calculator current prototype. It provides the calculation of all the required metrics that are used to generate the maintainability metric. Besides, it also provides exporting features to convenient formats like XML or JSON**.**
- **Structures:** This component contains the structures of the inputs and outputs models that the Metric Explorer uses. It also provides the functionality for generating the output file formats (XML, JSON).
- **Test Cases:** This component is provided for implementing the testing of the Metric Explorer component. It generates several use cases that test the functionality of the Maintainability Metric calculator. The test case generates console logs and XML files with several examples (DEWS and JavaPetStore).

5.2.3 Technical specifications

All the components are developed in JavaSE 1.6. So this is the minimum java version for executing the API. There are no any other requirements.

User Interface

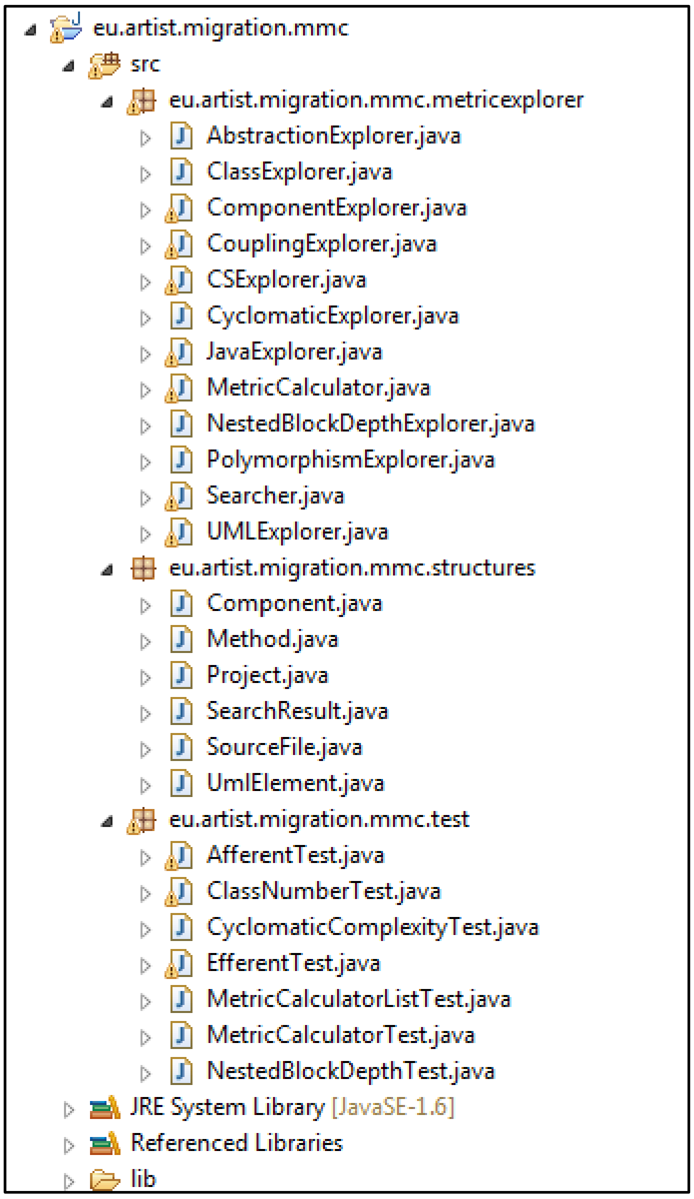
There is no user interface implemented as the result obtained from Maintainability Metric calculator will be consumed by other ARTIST tools or used for validation. For executing the Metrics Explorer API the user has to execute the test cases included in the API. Several input parameters can be changed for obtaining new metrics of different projects.

Back-end

There is no persistency implemented neither planned for this API. Thus, every time the new metrics are needed, they have to be calculated. Persistency is delegated to API consumers.

5.2.4 Package information

The following image depicts the package structure of the main component, the Artist metrics generator plug-in.



```
eu.artist.migration.mmc
    src
        eu.artist.migration.mmc.metricexplorer
            AbstractionExplorer.java
            ClassExplorer.java
            ComponentExplorer.java
            CouplingExplorer.java
            CSExplorer.java
            CyclomaticExplorer.java
            JavaExplorer.java
            MetricCalculator.java
            NestedBlockDepthExplorer.java
            PolymorphismExplorer.java
            Searcher.java
            UMLExplorer.java
        eu.artist.migration.mmc.structures
            Component.java
            Method.java
            Project.java
            SearchResult.java
            SourceFile.java
            UmlElement.java
        eu.artist.migration.mmc.test
            AfferentTest.java
            ClassNumberTest.java
            CyclomaticComplexityTest.java
            EfferentTest.java
            MetricCalculatorListTest.java
            MetricCalculatorTest.java
            NestedBlockDepthTest.java
    JRE System Library [JavaSE-1.6]
    Referenced Libraries
    lib
```

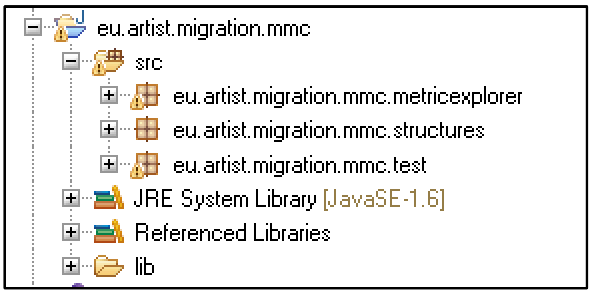
Figure 30 Package structure of the Maintainability metric calculator

- **eu.artist.migration.mmc.metricexplorer**: Contains the classes for exploring the UML models and the source code.
- **eu.artist.migration.mmc.strucctures**: Contains the classes of the structures used by the metric explorer component
- **eu.artist. migration.mmc.test**: Contains the test cases classes for executing the metric explorer component.

5.3 User Manual

In this version of the prototype this plug-in requires manual installation. The user has to import de components to the Eclipse workspace manually. All the components are developed in JavaSE 1.6. So this is the minimum java version for executing the API.

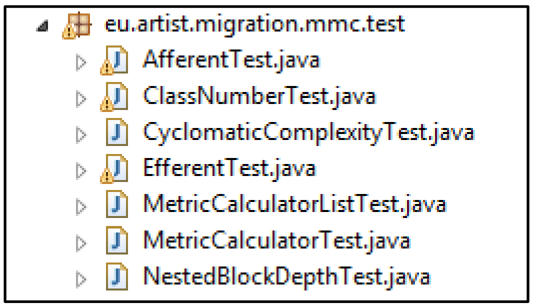
Import the project into the Eclipse workspace:



```
eu.artist.migration.mmc
  src
    eu.artist.migration.mmc.metricexplorer
    eu.artist.migration.mmc.structures
    eu.artist.migration.mmc.test
  JRE System Library [JavaSE-1.6]
  Referenced Libraries
  lib
```

Figure 31 Maintainability metric calculator project

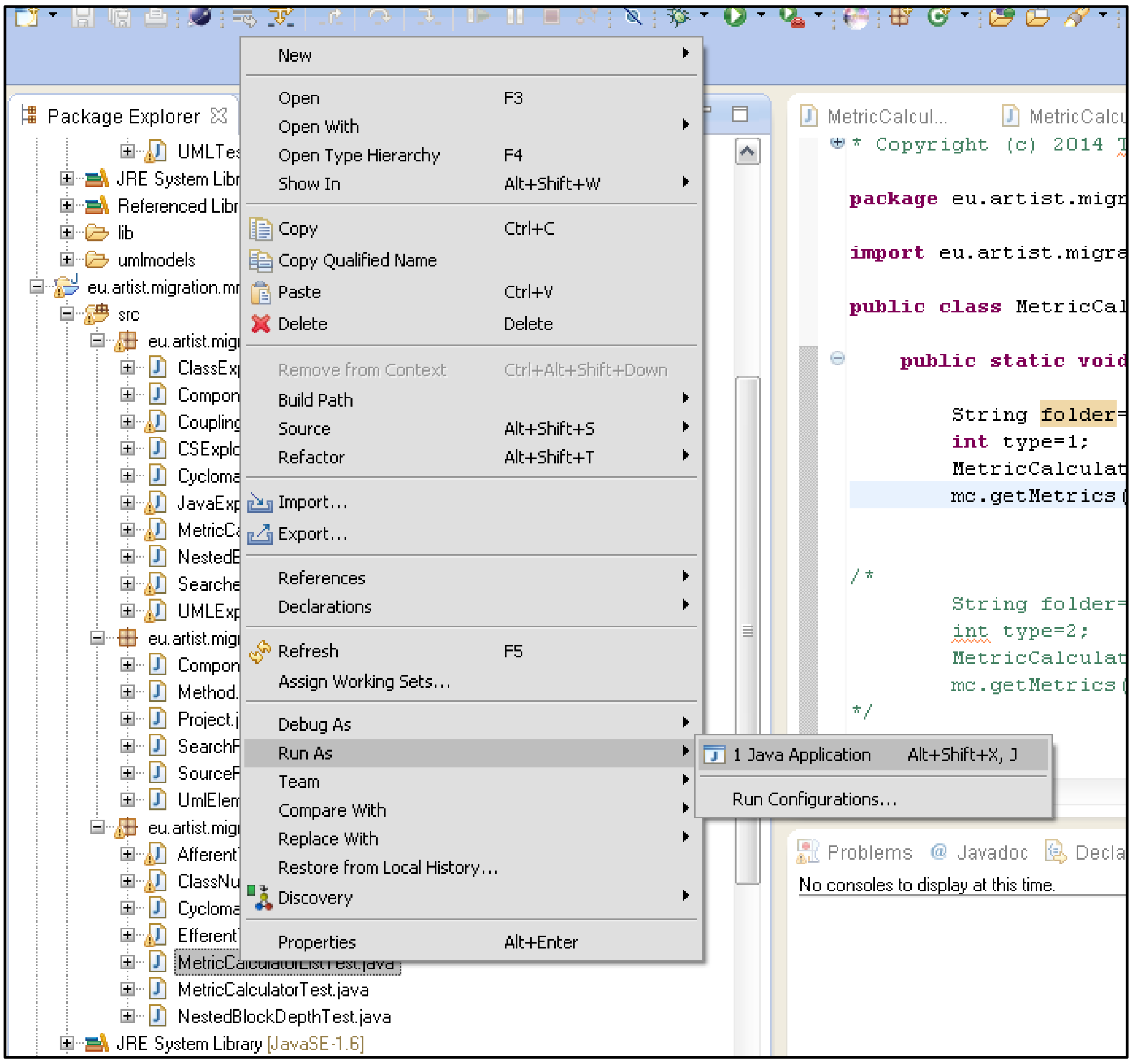
Open the eu.artist.migration.mmc.test package



```
eu.artist.migration.mmc.test
  AfferentTest.java
  ClassNumberTest.java
  CyclomaticComplexityTest.java
  EfferentTest.java
  MetricCalculatorListTest.java
  MetricCalculatorTest.java
  NestedBlockDepthTest.java
```

Figure 32 Testing package

Right click in a test case and select the Run as Java Application option:



```
* Copyright (c) 2014 T

package eu.artist.migr

import eu.artist.migra

public class MetricCal

    public static void

        String folder=
        int type=1;
        MetricCalculat
        mc.getMetrics(

/*
        String folder=
        int type=2;
        MetricCalculat
        mc.getMetrics(
*/
```

Figure 33 Maintainability metric calculator testing

***Note**: The user has to change the "hardcoded" input parameters of the test cases manually*

6 Delivery and Usage

6.1 Package Information

All realized components are packaged either as Eclipse plugins or Eclipse projects.

6.2 Installation Instructions

Components that have been realized in terms of Eclipse plugins need to be located in the respective *Eclipse plugins* folder. These components provide a dedicated UI to execute them. How they can be used is described in the respective user manuals.

Components that are not yet realized in terms of Eclipse plugins are provided as Eclipse projects. They need to be located in the respective Eclipse workspace to execute them. We have prepared a dedicated configuration to launch them in Eclipse. Again, how they can be launched is described in the respective user manuals.

6.3 User Manual

User manuals are provided by the sections describing the corresponding component.

6.4 Licensing Information

The selected license is the Eclipse Public License (EPL)[5] which is a known as a "commercial-friendly" open source license. This should facilitate the future potential reuse and integration of the toolbox (or at least of some of its components) by external partners.

6.5 Download Instructions

The sources of the different components developed in the context of this deliverable have been pushed to the public ARTIST repository:

```
https://github.com/artist-
project/ARTIST/tree/master/source/Tooling/migration/integrated environment
```

[5] „Eclipse Public License," [Online]. Available: http://www.eclipse.org/legal/epl-v10.html.

7 Conclusions

In this deliverable, we have reported on (i) the integration of ARTIST tools to support model-based software migration to the cloud, (ii) the development of a composition language to chain transformations as part of the concrete cloud-oriented migration scenario, (iii) advances in the development of transformations and profiles to realize cloud optimization patterns for a selected environment, and (iv) advances in the calculation of source-code metrics.

Considering the ARTIST integrated environment, it is developed on top of the Eclipse environment and targeted at software engineers and modellers to carry out cloud-oriented migration scenarios. The ARTIST integrated environment provides user interfaces, including tool bars, wizards, and views, for ARTIST tools that support such scenarios by a model-based approach.

As migration scenarios often require carrying out several different transformations to ultimately gain the expected software artefacts, we have developed a dedicated language that allows multiple transformations to be composed into a transformation chain, thereby enabling their automatic execution if the chain becomes executed. Transformations are loosely coupled by a more coarse-grained transformation, which fosters reuse while at the same time ensures that the software engineers and modellers still have the control over the execution of the single transformations.

Transformations developed in the context of the forward engineering phase of a migration scenario mainly refine and optimize models and application code towards a selected cloud environment. We have developed several transformations and profiles to support the refinement step with the main purpose to exploit novel cloud optimization opportunities. In fact, we have selected a set of optimization patterns contributed by WP9 and realized concrete transformations and profiles to support them by considering both the model level as well as the code level.

To measure the quality of the generated application code, we developed tool-support for calculating pertinent metrics, including maintainability, with the main purpose to reason about possible improvements achieved by a cloud-oriented software migration.

8 References

[1] Kusel, A., Schönböck, J., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: Reuse in Model-to-M
Languages: Are We There Yet? Software & Systems Modeling pp. 1–36 (2013)

[2] Bergmayr, A., Troya, J., Wimmer, M.: From out-place transformation evolution to in-place model patching. In
Automated Software Engineering (ASE). pp. 647– 652 (2014)

[3] Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Institut für Instrumentelle Mathematik, Bonn (1962)

[4] Bergmayr, A., Grossniklaus, M., Wimmer, M., Kappel, G.: „UML Profile Generation for Annotation-based M
Conf. on Softwae Engineering and Software Management (SE2015), 2015

[5] IEEE, "IEEE Standard Glossary of Software Engineering Terminology", 1990