ARTIST

FP7-317859

Advanced software-based seRvice provisioning and migraTIon of legacy Software

Deliverable D8.2.1

Components for Model Discovery from Legacy Technologies

Editor(s):	Hugo Bruneliere, Javier Canovas
Responsible Partner:	Inria
Status-Version:	Final - v1.0
Date:	26/09/2013
Distribution level (CO, PU):	PU

Project Number:	FP7-317859
Project Title:	ARTIST

Title of Deliverable:	Components for Model Discovery from Legacy Technologies
Due Date of Delivery to the EC:	30/09/2013

Workpackage responsible for the Deliverable:	WP8 – Legacy Product Analysis by Reverse Engineering
Editor(s):	Inria
Contributor(s):	Inria, ATOS, TUWien, Sparx, Spikes
Reviewer(s):	George Kousiouris (ICCS)
Approved by:	All Partners
Recommended/mandatory readers:	WP6, WP9, WP10

Abstract:	Concrete tool support allowing the generation of complete low-level models out of Java & C# legacy source artifacts, following the global methodology and recommendations (as provided in D8.1 or D8.2 for instance). The produced models will be then used as inputs to Model Understanding technologies (cf. D8.3.x)
Keyword List:	MDRE, Model Discovery, Discoverers
Licensing information:	This document itself is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/ The reported code is licensed under generally EPL

	(open source) for the Java EE support, indicated otherwise for the .NET support.

Document Description

Document Revision History

Version	*Date*	*Modifications Introduced*	
		Modification Reason	*Modified by*
v0.1	29/07/13	First draft version	Hugo Bruneliere (Inria)
v0.2	30/08/13	Second draft version (upgraded list of Model Discovery components and various corresponding editing)	Hugo Bruneliere (Inria)
v0.3	06/09/12	Master document (polishing list of Model Discovery components, adding missing figures, revising existing content)	Hugo Bruneliere (Inria)
v0.4	13/09/12	Version for internal review (integrating all the content from the various component providers)	Integration: Hugo Bruneliere (Inria) Contributions: Javier Canovas, Guillaume Doux and Matthieu Allon (Inria), Konrad Wieland (Sparx), Alexander Bergmayr (TUWien), Bram Pellens (Spikes)
V1.0	26/09/13	Complete version for submission to EC (integrating comments from internal review)	Hugo Bruneliere (Inria)

Table of Contents

Table of Figures

Table of Tables

Terms and abbreviations

API	Application Programming Interface
DLL	Dynamic-Link Library
EC	European Commission
EMF	Eclipse Modeling Framework
FE	Forward Engineering
IDE	Integrated Development Environment
KDM	Knowledge Discovery Metamodel
MDE	Model Driven Engineering
MDRE	Model Driven Reverse Engineering
PSM	Platform Specific Model
RE	Reverse Engineering
Sparx EA	Sparx Enterprise Architect
TS	Technical Space
UML	Unified Model Language
XML	Extensible Markup Language

Executive Summary

As ARTIST is dealing with the migration of legacy systems (to the Cloud), Reverse Engineering occupies a fundamental part of the developed methodology, general process and supporting tooling. In order to be able to efficiently realize both the migration feasibility analysis and actual migration itself, the prerequisite is to be able to obtain relevant and sufficient representations of the legacy system to be migrated.

More precisely, ARTIST is following a MDE-based approach were models are considered and used as main representations of a given legacy system. Thus, a Model Driven Reverse Engineering (MDRE) tooling is needed in order to (semi-)automatically produce these models out of the legacy artifacts composing the system. MDRE mainly consists in two main phases: Model Discovery and Model Understanding. While Model Discovery is about obtaining base (i.e., low-level) models from the various software artefacts composing the legacy system, the main purpose of Model Understanding is to perform a deeper analysis/interpretation of these discovered models to prepare and facilitate the next steps of the migration.

In the context of the ARTIST project, Task 8.2 is dedicated to the study, design and implementation of the support to this initial Model Discovery phase. The present document is focusing on the actual tooling support concretely implemented and provided within the context of ARTIST, namely the ARTIST Model Discovery Toolbox. The main objective of this toolbox is to provide to the migration engineers the set of generic components they require to build base models from the various legacy artifacts of a given legacy system. As part of ARTIST and related to the project's use cases, the support for both the Java EE and .NET technical spaces is being addressed. Thus, two main families of discovery components are provided accordingly. In addition to offering a big picture of the toolbox, this deliverable is also giving more insights on each one of these individual components developed by different partners in the project.

As mentioned previously, the models produced thanks to the ARTIST Model Discovery Toolbox are base models in the sense that they are direct/systematic representations of the content of the legacy artifacts, potentially conforming to different metamodels (e.g., UML, KDM or other more specific metamodels). It is very important to notice that no further analysis or interpretation of this content have been realized up to this point: the ARTIST Model Understanding Toolbox (cf. D8.3.1) is then in charge of performing these next operations.

1 Introduction

Within the context of ARTIST or more generally in any other software modernization project, different kinds of legacy artifacts very often have to be considered. They are usually heterogeneous in the sense of having different structures, formats, quality or relying on different technologies (cf. *D8.1 "Taxonomy of Legacy Artifacts"* for getting more details on these several aspects and their actual impact on Model Discovery).

The required first step of obtaining useful higher-level representations of such (legacy) software is called Reverse Engineering (RE), which is applied before Forward Engineering (FE) activities can be performed to actually obtain the modernized system. As promoted in the ARTIST approach, a Model Driven Reverse Engineering (MDRE) process is commonly composed of two main consecutive phases:

- **Model Discovery** which consists in (semi-)automatically producing initial base (i.e., low-level) and/or systematic models from the various artifacts composing the software system (e.g., source code, configuration files, data files, etc.);
- **Model Understanding** which allows computing, from the previously discovered models, a set of more specific models/views describing the system in such a way that the next Forward Engineering phase can be performed efficiently.

The present document and accompanying software actually focuses on the initial version (over three all along the project) of the concrete tool support for this first Model Discovery phase. This Model Discovery support will be conceptually detailed later on as part of upcoming *D8.2 "Methodologies and Techniques for Model Discovery"*. For your information, note that the corresponding Model Understanding support is described in *D8.3.1 "Mechanisms for Viewpoint Definition and View Extraction from Models of Legacy Artifacts"*.

The overall idea is not to provide a unique tool deeply integrating all the developed components, notably because of the technical heterogeneity of the legacy platforms (e.g., Java/J2EE vs. C#/.NET in the context of ARTIST). It is rather to offer an open Model Discovery Toolbox in which the migration engineer can pick up the components s/he needs to deal with this first phase of her/his MDRE process.

1.1 About this deliverable

This document is the complement to the delivered software components in the specified date and deliverable name at the head of the document.

1.2 Document structure

The document is structured as followed. A first part is an overview of the ARTIST Model Discovery Toolbox providing general information and presenting its architecture, main features and corresponding components. A second part is giving more detailed data about each one of the individual components (specific capabilities, technical implementation, etc.). A last part is offering more insights on the delivery and concrete usage of the toolbox, before the document is concluded and future works are proposed.

2 ARTIST Model Discovery Toolbox: Overview

2.1 Functional description

The purpose of this first version of the ARTIST Model Discovery Toolbox is to provide modernization engineers with useful base support for performing the mandatory Model Discovery step of their MDRE processes. The overall idea is, from the different legacy artifacts composing the legacy systems to be migrated, to obtain the required initial models for the next Model Understanding step to be performed efficiently (cf. deliverables D8.3.x).

Such a toolbox is fundamental in the ARTIST approach as the provision of (re)usable models out of the legacy system is a base prerequisite to the next steps of the model driven process: first Model Understanding as also part of Reverse Engineering (T8.3), then Forward Engineering (WP9). Obtaining sufficiently accurate models, even if not 100% complete, is also very important when dealing with the ARTIST pre-migration phase, notably with the modernization evaluation of effort and planning (WP5). In a more general context, the ARTIST Model Discovery toolbox can be practically useful in every activity implying the study or analysis of existing software systems in order to have a better vision of their current status.

The main innovation of the proposed toolbox is that it is open and extensible. The base support currently offered relies on a set of well-established IDEs and related capabilities: Eclipse Modeling project for the Java world and Sparx Enterprise Architect for .NET world. They have been/are being/will be reused and extended in order to design and build the new components implementing the required functionalities.

In term of concrete Model Discovery support, the features currently provided by this toolbox are the following (in italic the features already available but extended and/or reused in the project, in bold this first year newly developed features):

- From Java EE artifacts
  - From Java source code
    - *Java code model (complete)*
    - *KDM Code model (partial)*
    - ***Generic UML class diagram, structural part of the code (complete)*** – **direct** or *via KDM*
    - **Generic UML activity diagram, behavioural part of the code (in progress)**
    - *KDM Source model (artifacts inventory)*
  - From Java APIs/libraries (currently in Jars)
    - **Generic UML profile (in progress)**
- From .NET artifacts
  - From C# source code
    - *Generic UML class diagram, structural part of the code (complete)*
    - **Generic UML sequence diagram, at runtime execution (in progress)**
  - From Microsoft libraries (currently DLL files)
    - **Generic UML class diagram (in progress)**
  - From SharePoint files (XML-based, cf. also dedicated component below)
    - **SharePoint data model (in progress)**
- From XML-based artifacts
  - *Generic XML document model (complete)* + **XML model aggregator (partial)**

The coverage of the current support for Model Discovery (in terms of which types of software artefacts are currently possible to discover models from) can be analysed by using the dimensions identified in the taxonomy of legacy artefacts as proposed in ARTIST (cf. D8.1). Regarding the main axis of the taxonomy, the Technical Space (TS) dimension, which identifies a specific technological context (i.e., be defined as a working context with a set of associated concepts, body of knowledge, tools, required skills and capabilities), two main TSs are tackled: (1) *grammarware*, which includes those artefacts representing source code (i.e., Java and .NET source files) and (2) *xmlware*, which includes those artefacts defining XML data. It is also quite easy to analyse the dimension regarding the environment as the two main technologies are Java and .NET.

Concerning the other dimensions of the taxonomy, the legacy artifacts covered until now by the ARTIST Model Discovery Toolbox have the following characteristics:

- Origin – Most of the analyzed source code is manually produced, some configuration files are automatically initiated but then also completed manually;
- Purpose – All the treated artefacts are composing the actual source code of the legacy systems (no other inputs considered up to now);
- Consumer – The handled artefacts are generally intended to be processed by the system itself or its underlying infrastructure;
- Organization – Only well-structured artifacts (i.e., there is formalism to define the artefact structure such as a grammar for source code or an XML schema for XML documents) have been considered until now, less structured artefacts may be studied in the future if required by use cases;
- Nature – Purely static, the discovery from more dynamic artefacts has not been needed and/or experimented up to this point;
- Size – This is quite heterogeneous: the PetStore experiment (the practical case study developed for performing first experiments in ARTIST) is rather small and the tested use cases can be globally seen as having a medium/large size.
- Opacity – Only fully white-box artefacts have been analyzed until now (i.e., full access to the content of the legacy artefacts).
- Architectural layer – Different experiments have been conducted, but mainly the Data and Logic layers have been considered.

According to the future needs of the different ARTIST use cases, the coverage of the Model Discovery Toolbox regarding the taxonomy dimension (mainly Technical Space and Environment ones) could be extended in next versions.

2.2 Fitting into overall ARTIST solution

This tooling support for the Model Discovery step (ARTIST Task ID *MODELDISCO*) is inserted at the heart of the ARTIST approach and underlying modernization methodology. First, it is required for the ARTIST pre-migration phase: it allows obtaining base models of the legacy systems that can be used to evaluate the modernization effort, and so can potentially impact corresponding process planning. Then if the result of the pre-migration phase is positive, Model Discovery is the very first step of any actual ARTIST migration process, then followed by Model Understanding (ARTIST Task ID *MODELUNDER*, cf. also D8.3.x). Thus it is an important component for the Reverse Engineering step (WP8), as preceding the Forward Engineering one (WP9). Generally, the objective of this toolbox is to allow provisioning the remainder of the process with the sufficient and complete enough set of models, as discovered from the legacy artefacts composing the systems to be modernized. The expected support for both the Java and .NET worlds is shown in Figure 1.

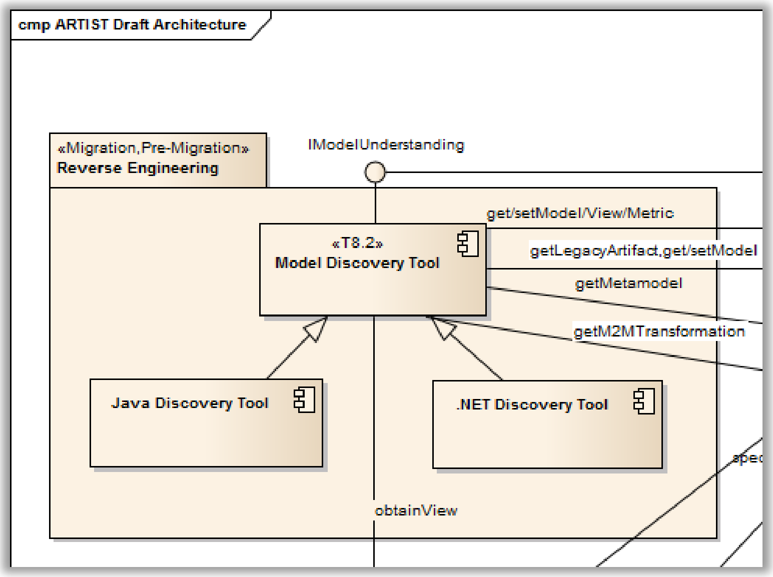


Figure 1 Excerpt of the ARTIST Architecture, the "Model Discovery" component

Going out of the ARTIST project, the Model Discovery toolbox has also value by itself. Independently from the considered global approach or process it may be used in, it can bring interesting benefits to all activities involving the simple observation or deeper analysis of existing software systems.

2.3 Technical description

From an implementation perspective, the choice has been made very early to rely as much as possible on already existing technologies:

- Eclipse EMF [2], MoDisco [4] & ATL [1] for the Java EE world;
- Sparx Enterprise Architect [12] for the .NET world.

The objective is to be able to benefit from the well-established tool support and related communities. The plan is of course to reuse the various provided components as a base for the Model Discovery Toolbox developments, but also to potentially upgrade/extend them if needed or relevant within the context of the project (e.g., for the sake of developing new model discoverers targeting types of legacy artifacts that are not supported yet).

As a model-based solution, the Model Discovery Toolbox is built upon a set of reference metamodels. Notably, the OMG UML standard metamodel [10] is used as the base for models produced as outputs of the toolbox while the OMG KDM standard metamodel [9] is also considered as an alternative option for interoperability purposes. For the Java world, all metamodels are actually implemented in EMF Ecore: the UML metamodel implementation is provided by the Eclipse UML2 project [6] whereas the Java, KDM and XML ones are offered by the Eclipse MoDisco project [4]. For the .NET world, the UML2 metamodel implementation is the one fitted with Sparx Enterprise Architecture [12]. Importantly, an additional EA support for importing/exporting EMF models from/to Eclipse is being developed as part of ARTIST in order to allow model interchange between the two worlds.

According to the current list of features as presented in Section 2.1, here is some more information about the technical implementation of each one of them (in italic the features already available but extended and/or reused in the project, in bold this first year newly developed features):

- From Java EE artifacts
    - From Java source code
        - *Java code model (complete)*
            - MoDisco base support reused as is in the project – implementation in Java
        - *KDM code model (partial)*
            - MoDisco base support reused as is in the project – implementation in Java
        - ***Generic UML class diagram, structural part of the code (complete)*** – **direct** or *via KDM*
            - MoDisco version using the "Java to KDM to UML" transformation chain – implementation in Java + ATL
            - Newly developed version (already quite stable) directly from the Java code model – implementation in ATL
        - **Generic UML activity diagram, behavioural part of the code (in progress)**
            - Newly developed version directly from the Java code model – implementation in ATL
        - *KDM Source model (artifacts inventory)*
            - MoDisco base support reused as is in the project – implementation in Java
    - From Java APIs/libraries (currently in Jars)
        - **Generic UML profile (in progress)**
            - Newly developed prototype – implementation in Java
- From .NET artifacts
    - From C# source code
        - *Generic UML class diagram, structural part of the code (complete)*
            - Based on existing support, it has been extended and specialized – implementation in C#/.NET
        - **Generic UML sequence diagram, at runtime execution (in progress)**
            - Newly developed Enterprise Architecture extension – implementation in C#/.NET
    - From Microsoft libraries (currently DLL files)
        - **Generic UML class diagram (in progress)**
            - Newly developed Enterprise Architecture extension – implementation in C#/.NET
    - From SharePoint files (XML-based, cf. also dedicated component below)
        - **SharePoint data model (in progress)**
            - Newly developed prototype – implementation using the generic XML discovery support + transformations in ATL
- From XML-based artifacts
    - *Generic XML document model (complete)* + **XML model aggregator (partial)**
        - MoDisco base support reused and extended in the project – implementation in Java

2.4 Toolbox global architecture

From an architectural perspective, the ARTIST Model Discovery Toolbox is globally composed of two main entities: the Java support (Eclipse-based) and the .NET support (Sparx EA-based), as shown on Figure 2.

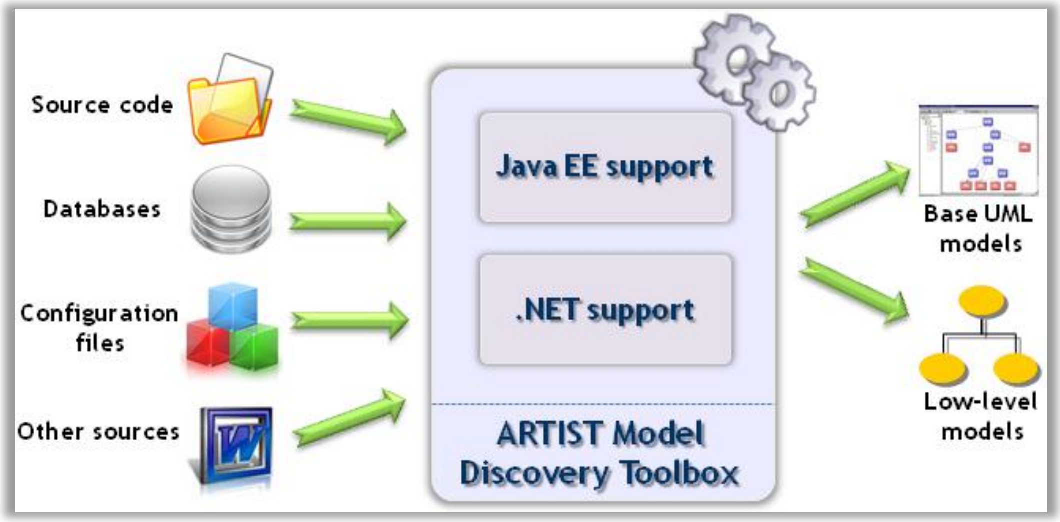


Figure 2 ARTIST Model Discovery Toolbox: Overall Architecture

The Model Discovery toolbox is taking as inputs different kinds of legacy artifacts from both the Java EE and .NET worlds, and producing as outputs base models that can conform to UML or to (more specific) low-level metamodels. These models are so-called Platform-Specific Models (or PSM) according to the common MDE jargon and are then used as inputs of the ARTIST Model Understanding Toolbox (cf. D8.3.1).

For the Java support, the integration of the different components (e.g., model discoverers) is realized notably via the Eclipse MoDisco framework. As dedicated to MDRE activities and intensively reused/enhanced in the context of ARTIST Reverse Engineering tasks, it is a perfect candidate. For the .NET world, a similar integration is performed via Sparx Enterprise Architect (components are actually Sparx EA add-ons), as a tool suite developed by one of the partners of the project.

Next section presents with more details each one of the different components forming these two main families of support, respectively for the Java EE and .NET (legacy) technologies.

2.5 General Delivery Information

The sources of the different components newly developed during this first year of the ARTIST project have been grouped in a Zip file that has been put on the ARTIST (Livelink) repository at Deliverables/M12 folder.

Concerning the other components that are used and sometimes improved in ARTIST but were already existing prior to the project (e.g., from MoDisco [4] or Sparx EA [12]), they generally remain available from their respective infrastructures and continue to follow their regular delivery cycles.

For more installation details or user manual information on each individual component, please refer to the corresponding describing sections (as presented after in Section 3 of this document).

2.6 General Licensing Information

Generally, the objective is to propose the Model Discovery Toolbox as an open solution resulting from the ARTIST project. However, according to the nature of the two mainly targeted worlds/markets (i.e., Java and .NET), the overall approach is either fully open source or more mixed.

For all the Java/Eclipse based components (e.g., for the Java EE support mainly), 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.

For the Sparx Enterprise Architect-based components (e.g., for the .NET support mainly), the selected license for most of the new developments will also be open source. However, the corresponding components require Sparx EA to be installed, which is released under a proprietary license according to the policy explained below:

- **Free Trial -** Enterprise Architect offers a free trial version, which provides full functionality for 30 days. All results can be used with any other license type as well.
- **Academic License (Named User and Floating) -** Enterprise Architect academic program offers students single-user licenses, educational classroom licenses, up to a campus licenses (EA Ultimate Edition). The academic licensing course includes the same functionality as commercial products. Under the Sparx Systems academic program, students can purchase individual Enterprise Architect licenses and EA add-ins at academic pricing levels. This program may be used exclusively for non-commercial purposes and is only available for lecturers, teachers, pupils, students or trainers.
- **Commercial License (Named User and Floating) -** Enterprise Architect provides, apart from the free trial and the academic licenses, commercial licenses. These can be obtained in different variants ranging from Desktop license to the Ultimate license. Starting from the Corporate license unnamed licenses are available. All working products, originating from Enterprise Architect, can be transformed to Eclipse. Thus, they are available either within Enterprise Architect or in other environments.

For more specific licensing details on each individual component, please refer to the corresponding describing section (as presented previously in this document).

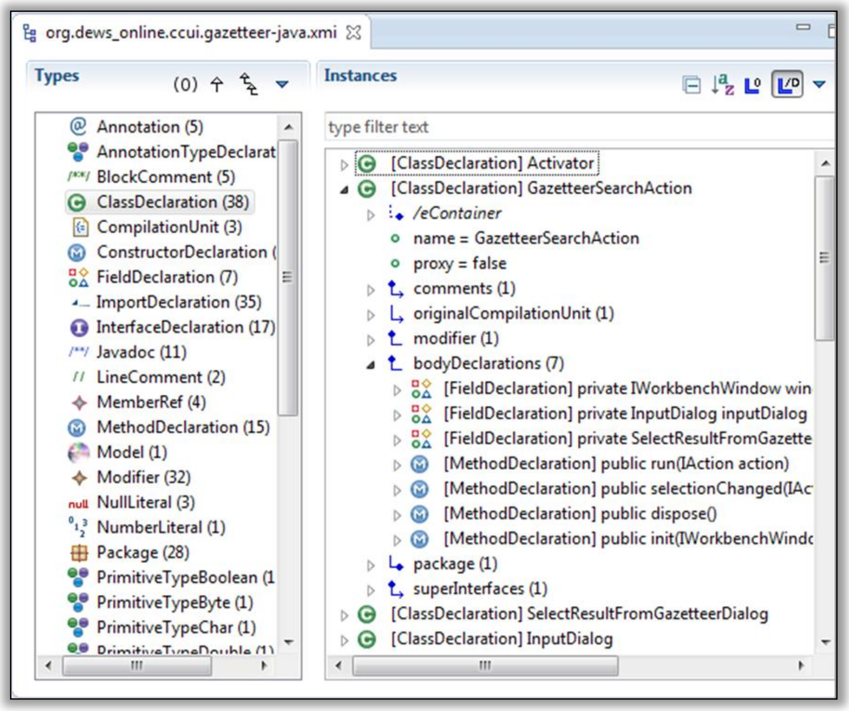
3 ARTIST Model Discovery Toolbox: Components Details

As introduced before in the overview section, the toolbox is actually made of a set of components covering different legacy technologies. In the context of ARTIST, the two primarily addressed ones are Java EE and .NET. However, other more general components can also be added in the toolbox when relevant. This section is about giving some more details on each one of the included components.

3.1 Java EE Support

3.1.1 Component "Java Code Model Discoverer from Java source code"

This component offers both a full Java metamodel (based on the Java Language Specification) and the corresponding model discoverer. The latter allows obtaining complete Java models out of any Java source code. Depending on the entered parameters, the Java model can represent every data available in the initial source code, from its superstructure (packages, interfaces, classes, field and method declarations, etc.) to the full details on the expressions and statements (e.g., inside method bodies).



```
Types
Annotation (5)
AnnotationTypeDeclarat
BlockComment (5)
ClassDeclaration (38)
CompilationUnit (3)
ConstructorDeclaration (
FieldDeclaration (7)
ImportDeclaration (35)
InterfaceDeclaration (17)
Javadoc (11)
LineComment (2)
MemberRef (4)
MethodDeclaration (15)
Model (1)
Modifier (32)
NullLiteral (3)
NumberLiteral (1)
Package (28)
PrimitiveTypeBoolean (1
PrimitiveTypeByte (1)
PrimitiveTypeChar (1)
```

```
Instances
[ClassDeclaration] Activator
[ClassDeclaration] GazetteerSearchAction
    /eContainer
    name = GazetteerSearchAction
    proxy = false
    comments (1)
    originalCompilationUnit (1)
    modifier (1)
    bodyDeclarations (7)
        [FieldDeclaration] private IWorkbenchWindow win
        [FieldDeclaration] private InputDialog inputDialog
        [FieldDeclaration] private SelectResultFromGazette
        [MethodDeclaration] public run(IAction action)
        [MethodDeclaration] public selectionChanged(IAc
        [MethodDeclaration] public dispose()
        [MethodDeclaration] public init(IWorkbenchWindc
    package (1)
    superInterfaces (1)
[ClassDeclaration] SelectResultFromGazetteerDialog
[ClassDeclaration] InputDialog
```

Figure 3 Java Code model opened in the MoDisco Model Browser

The component is provided by the Eclipse MoDisco project (initiated and created within the context of the IST-FP6 MODELPLEX EU project). However, it is practically reused in ARTIST (as an individual component or to provide inputs to some newly developed components) and so is considered as a base component from the ARTIST Model Discovery Toolbox.

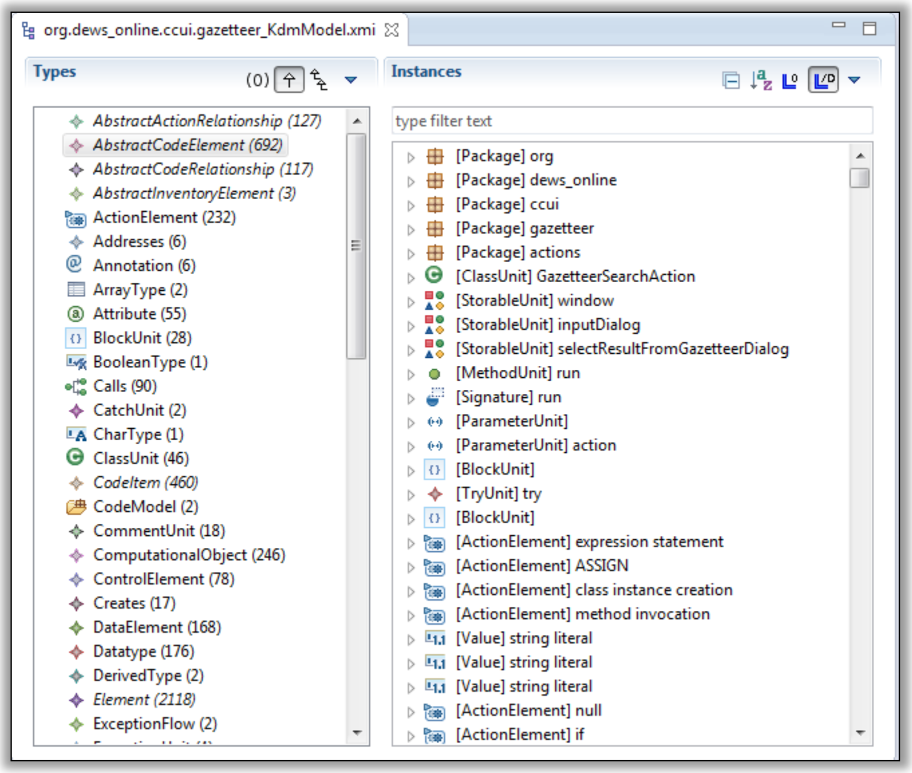
This particular component is implemented as Eclipse plugins using Java, JDT (notably the parsing capabilities) [3] and being based on EMF [2]. It is open source and available under the Eclipse Public License (EPL) [5].

For more information, reference documentation or download/installation instructions, please go to the MoDisco homepage [4].

3.1.2 Component "KDM Code Model Discoverer from Java source code"

This component is based on the chaining of the previous Java Code Model Discoverer with a Java-to-KDM model transformation. It also uses an Ecore implementation of the OMG KDM metamodel. Basically, this component allows producing a KDM Code model from initial Java source code (passing by an intermediate Java source code model). It is important to note that, contrary to the Java source code model, the obtained KDM Code model only covers the superstructure of the program and not the details of the expressions and statements (method bodies).

In the ARTIST context and also more generally, such a KDM support is interesting as keeping the opportunity for interoperability with other tools or solutions relying on OMG ADM standards [7].


org.dews_online.ccui.gazetteer_KdmModel.xmi

Types (0)

AbstractActionRelationship (127)
AbstractCodeElement (692)
AbstractCodeRelationship (117)
AbstractInventoryElement (3)
ActionElement (232)
Addresses (6)
Annotation (6)
ArrayType (2)
Attribute (55)
BlockUnit (28)
BooleanType (1)
Calls (90)
CatchUnit (2)
CharType (1)
ClassUnit (46)
CodeItem (460)
CodeModel (2)
CommentUnit (18)
ComputationalObject (246)
ControlElement (78)
Creates (17)
DataElement (168)
Datatype (176)
DerivedType (2)
Element (2118)
ExceptionFlow (2)

Instances

type filter text

[Package] org
[Package] dews_online
[Package] ccui
[Package] gazetteer
[Package] actions
[ClassUnit] GazetteerSearchAction
[StorableUnit] window
[StorableUnit] inputDialog
[StorableUnit] selectResultFromGazetteerDialog
[MethodUnit] run
[Signature] run
[ParameterUnit]
[ParameterUnit] action
[BlockUnit]
[TryUnit] try
[BlockUnit]
[ActionElement] expression statement
[ActionElement] ASSIGN
[ActionElement] class instance creation
[ActionElement] method invocation
[Value] string literal
[Value] string literal
[Value] string literal
[ActionElement] null
[ActionElement] if

Figure 4 KDM Code model opened in the MoDisco Model Browser

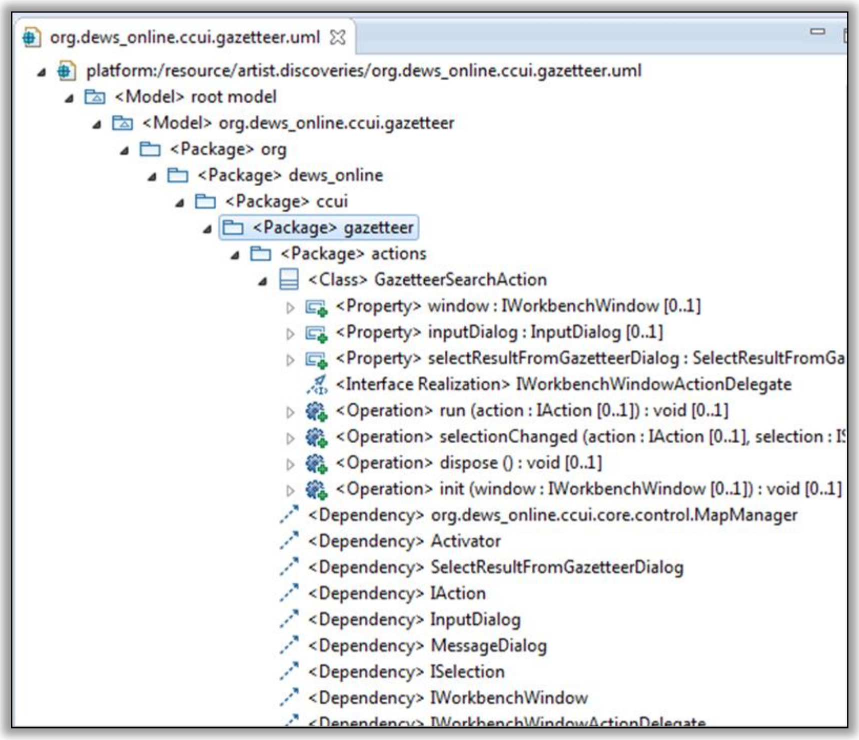
The component is provided by the Eclipse MoDisco project (initiated and created within the context of the IST-FP6 MODELPLEX EU project). However, it is practically reused in ARTIST (as an individual component or to provide inputs to some newly developed components) and so is considered as a base component from the ARTIST Model Discovery Toolbox.

As introduced before, this particular component is actually made of a Java-to-KDM model transformation implemented in Java-EMF [2] and is packaged as Eclipse plugins. It is open source and available under the Eclipse Public License (EPL) [5].

For more information, reference documentation or download/installation instructions, please go to the MoDisco homepage [4].

3.1.3 Component "Generic UML Class Diagram Discoverer (structural part of the code) from Java source code – via KDM"

This component is based on the chaining of the previous KDM Code Model Discoverer with a KDM-to-UML model transformation. It also uses Ecore implementations of the OMG KDM and UML metamodels. Basically, this component allows producing a UML Class Diagram model from initial Java source code (passing by intermediate Java source code and KDM Code models). It is important to note that, similarly to the KDM Code model, the obtained UML Class Diagram model only covers the structure of the program and not the details of the expressions and statements (method bodies). These behavioural aspects are to be actually covered by the Generic UML Activity Diagram Model Discoverer (cf. section 3.1.5).



```
org.dews_online.ccui.gazetteer.uml
platform:/resource/artist.discoveries/org.dews_online.ccui.gazetteer.uml
  <Model> root model
    <Model> org.dews_online.ccui.gazetteer
      <Package> org
        <Package> dews_online
          <Package> ccui
            <Package> gazetteer
              <Package> actions
                <Class> GazetteerSearchAction
                  <Property> window : IWorkbenchWindow [0..1]
                  <Property> inputDialog : InputDialog [0..1]
                  <Property> selectResultFromGazetteerDialog : SelectResultFromGa
                  <Interface Realization> IWorkbenchWindowActionDelegate
                  <Operation> run (action : IAction [0..1]) : void [0..1]
                  <Operation> selectionChanged (action : IAction [0..1], selection : IS
                  <Operation> dispose () : void [0..1]
                  <Operation> init (window : IWorkbenchWindow [0..1]) : void [0..1]
                <Dependency> org.dews_online.ccui.core.control.MapManager
                <Dependency> Activator
                <Dependency> SelectResultFromGazetteerDialog
                <Dependency> IAction
                <Dependency> InputDialog
                <Dependency> MessageDialog
                <Dependency> ISelection
                <Dependency> IWorkbenchWindow
                <Dependency> IWorkbenchWindowActionDelegate
```

Figure 5 UML Class Diagram model opened in the MoDisco Model Browser

The component is provided by the Eclipse MoDisco project (initiated and created within the context of the IST-FP6 MODELPLEX EU project). However, it is practically reused in ARTIST (as an individual component or to provide inputs to some newly developed components) and so is considered as a base component from the ARTIST Model Discovery Toolbox.

As introduced before, this particular component is actually made of a KDM-to-UML model transformation implemented in ATL [1] and is packaged as Eclipse plugins. It is open source and available under the Eclipse Public License (EPL) [5].

For more information, reference documentation or download/installation instructions, please go to the MoDisco homepage [4].

3.1.4 Component "Generic UML Class Diagram Discoverer (structural part of the code) from Java source code – direct"

3.1.4.1 Overview

This new component, packaged as a MoDisco discoverer, is based on the chaining of the Java Code Model Discoverer with a newly developed Java-to-UML (Class Diagram) model transformation implemented in ATL [1]. It also relies on an Ecore implementation of the OMG UML metamodel [10]. Similarly to the previous component, it allows producing an UML Class Diagram model from initial Java source code (but this time only passing by an intermediate Java source code model). The obtained UML Class Diagram model also only covers the superstructure of the program and not the details of the expressions and statements (method bodies). These behavioural aspects are to be actually covered by the Generic UML Activity Diagram Model Discoverer (cf. section 3.1.5).

3.1.4.2 Functional description

This component currently considers:

- As input: Java source code, or eventually the Java model representing the Java source code (if the Java Code Model Discoverer has already been applied separately before).
- As output: a UML model containing the UML class diagram generated out of the input.

The intermediate Java code model being a direct representation of the content of the initial Java sources, the actual conceptual mapping between Java and UML (Class Diagram) is specified by the Java-to-UML model transformation as previously introduced.

An overview (non-exhaustive) of this mapping between elements from these two metamodels is presented in Table 1.

Table 1 Mapping (excerpt) used in the Java-to-UML Class diagram transformation

JAVA metamodel	**UML metamodel (Class Diagram)**
Model	Model
Package	Package
[ClassDeclaration OR NamedElement] **(1)**	Class
[UnresolvedItem OR UnresolvedTypeDeclaration OR InterfaceDeclaration] **(2)**	Interface
TypeAccess	[Generalization OR InterfaceRealization] **(3)**
FieldDeclaration	[Property OR Association] **(4)**
MethodDeclaration	Operation
SingleVariableDeclaration	Parameter
PrimitiveType	PrimitiveType

(1) *: The use of 'NamedElement' happens only when elements are unresolved (in the Java code model).*

(2) *: The two first types concern unresolved items (in the Java code model) which are not used in a class as a type or in class imports.*

(3) *: This special case occurs when we have an interface inheriting from another.*

(4) *: In the ATL transformation, we need the 'FieldDeclaration' type to indicate (in the created association) which fields are related by the association.*

3.1.4.3 Technical specifications

The Java-to-UML model transformation has been developed and is being executed thanks to the ATL tooling [1]. The implemented ATL artefacts are the following ones*:*

- An ATL module where all the transformation rules between the Java elements and the UML ones are defined.
- An ATL library containing all the helpers (i.e., reusable methods in the ATL jargon) used by the different rules in the ATL module.

This transformation has been embedded in and packaged as a MoDisco Discoverer (currently named *fr.inria.artist.modisco.java.uml.discoverer.java2uml*). This discoverer allows the selection of a Java code model, and automatically sets the configuration needed to launch the transformation execution and serialization of the produced UML model.

As a summary, here are the concrete technologies that have been used and deployed in the context of this component:

- Java metamodel and Java model discovery – Eclipse MoDisco [4]
- UML metamodel – Eclipse UML2 [6]
- (Java-to-UML) model transformation – Eclipse ATL [1]

General information:

- Status: Alpha
- Release: 0.1

3.1.4.4 Download and installation instructions

Thus, this component requires the following software to be installed:

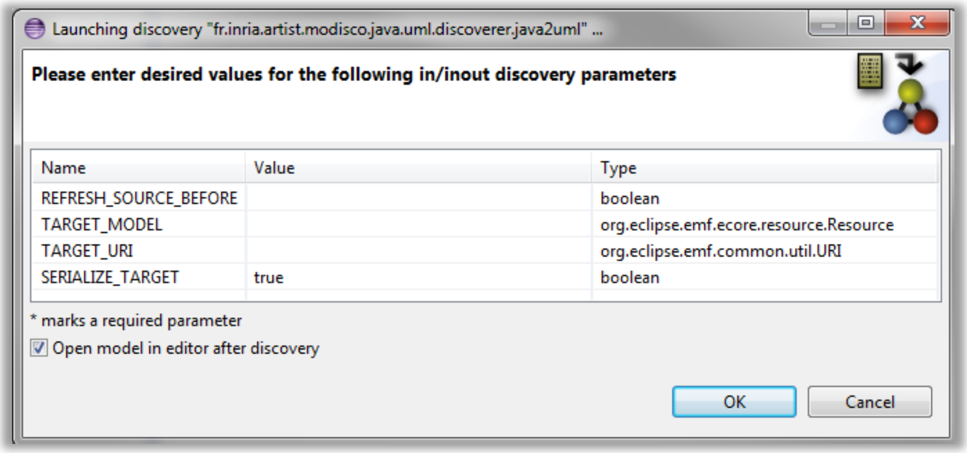
- Eclipse Platform (get the latest Eclipse Modeling bundle)
- Eclipse MoDisco SDK
- Eclipse ATL SDK
- Eclipse UML2

As for the other newly developed components, the sources can be downloaded from the public repository (cf. section 2.5).

3.1.4.5 User manual

To use the component, you can start from a Java code model, as previously obtained from the Java Code Model Discoverer for instance (cf. section 3.1.1), and then:

- Right-click on the Java code model.
- Select the following menus *"Discovery / Discoverers / Java2UMLDiscoverer – Class Diagram"*.
- Enter, as shown on Figure 6, whether you want to serialize the output UML model and manually set its location.
- Validate your choices in the window, and run the discovery process. The resulting UML model will by default (i.e., if not specified differently) be saved at the same level than the Java code model.



Launching discovery "fr.inria.artist.modisco.java.uml.discoverer.java2uml" ...

Please enter desired values for the following in/inout discovery parameters

Name	Value	Type
REFRESH_SOURCE_BEFORE		boolean
TARGET_MODEL		org.eclipse.emf.ecore.resource.Resource
TARGET_URI		org.eclipse.emf.common.util.URI
SERIALIZE_TARGET	true	boolean

* marks a required parameter

[✓] Open model in editor after discovery

Figure 6 Parameter selection for the UML Class Diagram Model Discoverer

The obtained UML model can finally be navigated using the MoDisco Model Browser for instance.

3.1.4.6 *Licensing information*

This component is provided as open source under the Eclipse Public License [5].

3.1.5 Component "Generic UML Activity Diagram Discoverer (behavioural part of the code) from Java source code – direct"

3.1.5.1 *Overview*

This new component, packaged as a MoDisco discoverer, is based on the chaining of the Java Code Model Discoverer with a newly developed Java-to-UML (Activity Diagram) model transformation implemented in ATL [1]. It also relies on an Ecore implementation of the OMG UML metamodel [10]. It allows producing an UML Activity Diagram model from initial Java source code (only passing by an intermediate Java source code model). Complementary to UML Class Diagram models produced by other components, the obtained UML Activity Diagram model covers the behavioural aspects of the initial Java code by detailing the expressions and statements (as expressed inside method bodies).

3.1.5.2 *Functional description*

This component currently considers:

- As input: Java source code, or eventually the Java model representing the Java source code (if the Java Code Model Discoverer has already been applied separately before).
- As output: a UML model containing UML activity diagrams corresponding to each one of the methods represented in the Java code model.

The intermediate Java code model being a direct representation of the content of the initial Java sources, the actual conceptual mapping between Java and UML (Activity Diagram) is specified by the Java-to-UML model transformation as previously introduced.

An overview (non-exhaustive) of this mapping between elements from these two metamodels is presented in Table 2.

JAVA metamodel	UML metamodel (Activity Diagram)
Model	Model, containing the activities Package, containing the classes needed to type the activities
Method	Activity, containing the parameters, body, return expression and a result node
SingleVariableDeclaration ***(1)***	ActivityParameterNode, linked to a ForkNode by an ObjectFlow
Block	SructuredActivityNode, containing an Initial and Final Nodes and the mapping of the Statements from the Block
ReturnStatement	StructuredActivityNode, containing the mapping of the return expression and linking the output of this one to the result node of the Activity
[Statement AND Expression]	Complex patterns ***(2)***
PrimitiveType	PrimitiveType
ClassDeclaration	Class
InterfaceDeclaration	Interface

Table 2 Mapping (excerpt) used in the Java-to-UML Activity diagrams transformation

(1) *: For the methods parameter representation.*
(2) *: See the FUML specification [7].*

3.1.5.3 Technical specifications

The technical aspects of this component are very similar to the ones of the previous UML Class Diagram Model Discoverer. It relies on a main ATL transformation (module) and has been packaged as a MoDisco discoverer in a same way (named *fr.inria.artist.modisco.java.uml .discoverer.java2uml.activity.diagram*). The Eclipse MoDisco [4] and Eclipse ATL [1] technologies have also been used as a base for implementing this component.

This discoverer allows the selection of a Java code model, and automatically sets the configuration needed to launch the transformation execution and serialization of the produced UML model.

General information:

- Status: Alpha
- Release: 0.1

3.1.5.4 Download and installation instructions

Thus, this component requires the following software to be installed:

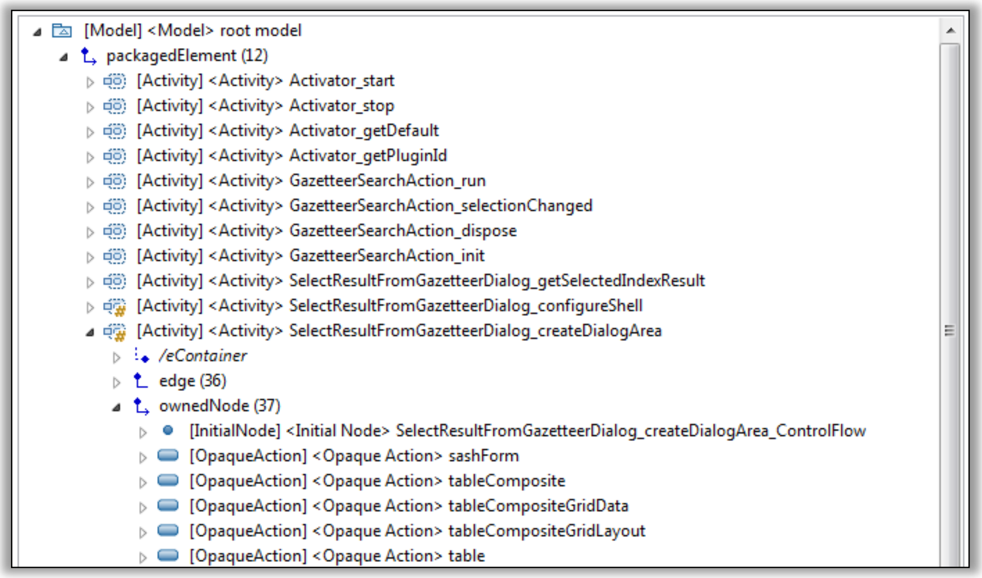
- Eclipse Platform (get the latest Eclipse Modeling bundle)
- Eclipse MoDisco SDK
- Eclipse ATL SDK
- Eclipse UML2

As for the other newly developed components, the sources can be downloaded from the public repository (cf. section 2.5).

3.1.5.5 User manual

To use the component, you can start from a Java code model, as previously obtained from the Java Code Model Discoverer for instance (cf. section 3.1.1), and then:

- Right-click on the Java code model.
- Select the following menus *"Discovery / Discoverers / Java2UMLDiscoverer – Activity Diagram"*.
- Enter whether you want to serialize the output UML model and manually set its location.
- Validate your choices in the window, and run the discovery process. The resulting UML model will by default (i.e., if not specified differently) be saved at the same level than the Java code model.



```
[Model] <Model> root model
  packagedElement (12)
    [Activity] <Activity> Activator_start
    [Activity] <Activity> Activator_stop
    [Activity] <Activity> Activator_getDefault
    [Activity] <Activity> Activator_getPluginId
    [Activity] <Activity> GazetteerSearchAction_run
    [Activity] <Activity> GazetteerSearchAction_selectionChanged
    [Activity] <Activity> GazetteerSearchAction_dispose
    [Activity] <Activity> GazetteerSearchAction_init
    [Activity] <Activity> SelectResultFromGazetteerDialog_getSelectedIndexResult
    [Activity] <Activity> SelectResultFromGazetteerDialog_configureShell
    [Activity] <Activity> SelectResultFromGazetteerDialog_createDialogArea
      /eContainer
      edge (36)
      ownedNode (37)
        [InitialNode] <Initial Node> SelectResultFromGazetteerDialog_createDialogArea_ControlFlow
        [OpaqueAction] <Opaque Action> sashForm
        [OpaqueAction] <Opaque Action> tableComposite
        [OpaqueAction] <Opaque Action> tableCompositeGridData
        [OpaqueAction] <Opaque Action> tableCompositeGridLayout
        [OpaqueAction] <Opaque Action> table
```

Figure 7 UML Activity Diagram model opened in the MoDisco Model Browser

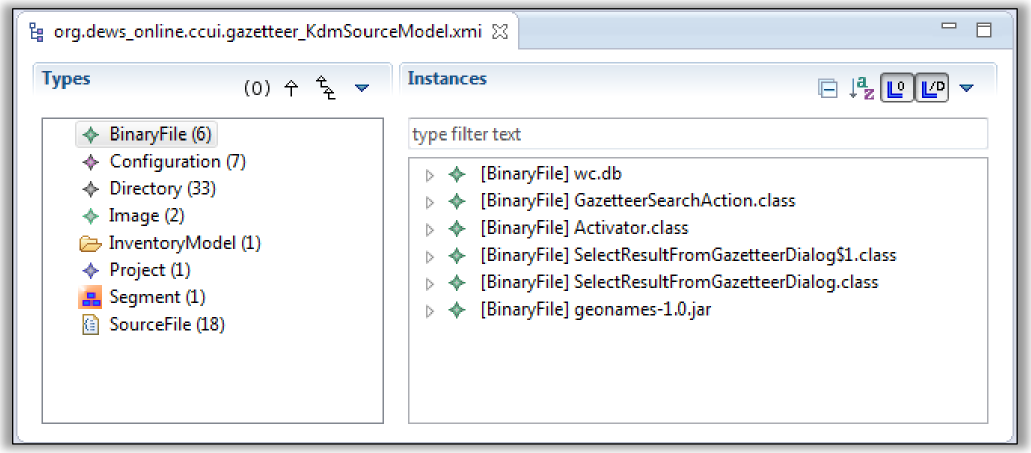
The obtained UML model can finally be navigated using the MoDisco Model Browser for instance (as shown on Figure 7).

3.1.5.6 Licensing information

This component is provided as open source under the Eclipse Public License [5].

3.1.6 Component "KDM Source Model Discoverer from Java project"

This component allows obtaining KDM Source model out of any Java project, using an Ecore implementation of the OMG KDM metamodel. Basically, this KDM Source model represents the different software entities composing the Java projects (directories, source files, binary files, configuration files, etc.). It is important to note that, contrary to the KDM Code model, the obtained model is more high-level and does not provide details on the insights of the described artefacts.


org.dews_online.ccui.gazetteer_KdmSourceModel.xmi

Types (0)

- BinaryFile (6)
- Configuration (7)
- Directory (33)
- Image (2)
- InventoryModel (1)
- Project (1)
- Segment (1)
- SourceFile (18)

Instances

type filter text

- [BinaryFile] wc.db
- [BinaryFile] GazetteerSearchAction.class
- [BinaryFile] Activator.class
- [BinaryFile] SelectResultFromGazetteerDialog$1.class
- [BinaryFile] SelectResultFromGazetteerDialog.class
- [BinaryFile] geonames-1.0.jar

Figure 8 KDM Source model opened in the MoDisco Model Browser

The component is provided by the Eclipse MoDisco project (initiated and created within the context of the IST-FP6 MODELPLEX EU project). However, it is practically reused in ARTIST (as an individual component or to provide inputs to some newly developed components) and so is considered as a base component from the ARTIST Model Discovery Toolbox.

This particular component is implemented as Eclipse plugins using Java, JDT (notably the parsing capabilities) [3] and being based on EMF [2]. It is open source and available under the Eclipse Public License (EPL) [5].

For more information, reference documentation or download/installation instructions, please go to the MoDisco homepage [4].

3.1.7 Component "Generic UML Profile Discoverer from Java API/library"

3.1.7.1 Overview

Most modern programming languages nowadays provide a mechanism to express attributes over program elements, which lead to what is called attribute-oriented programming. Utilizing meta-attributes is widely adopted in practice and supported in several programming languages through, for example, annotations in Java and Scala, attributes in C#, and decorators in Python. The focus of the *Profile Discoverer* component is on Java annotations as Java already introduced the concept in 2004 and, therefore, many known libraries have embraced them, such as the Java Persistence API (JPA) to denote strong and weak entities, Enterprise Java Beans (EJB) to manage the state of session beans, Spring to provide inversion of control and dependency injection, and Guava to define non-nullness for method parameters and return values.

The practical value of attributes has not only been recognized at the programming level but also at the modelling level. Notably, in UML, stereotypes show similar representational capabilities as annotations in Java. Therefore, UML stereotypes can be derived from established libraries and frameworks in order to produce corresponding UML Profiles (cf. example from Figure 9). The availability of UML Profiles ensures that annotations in the legacy software can also be represented at the modelling level. In D8.3.1, the application of UML stereotypes throughout the reverse-engineering of UML models from Java code is demonstrated. From the perspective of the forward-engineering, UML stereotypes can be used to render UML models more specific.

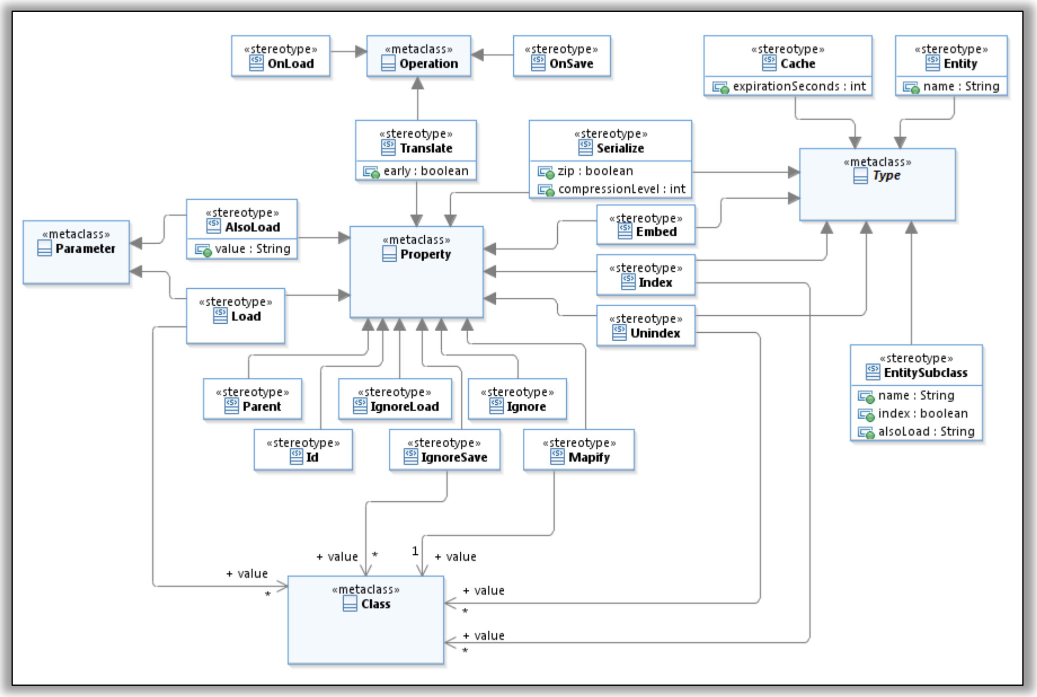


Figure 9 UML Profile model discovered from a Java library (*jar* file)

3.1.7.2 Functional Description

The entry-point for the UML Profile discovery, as depicted in Figure 10, is *Java Code* that is converted into a corresponding *Code Model* thanks to the previously presented Java Code Model Discoverer. *Code Models* are the basis for discovering UML Profiles, as depicted in (2a). They serve as foundation for annotating UML models with the same meta-information as Java code elements. Before such UML stereotypes can be applied, they need to be defined. As shown in (2b), such a definition produces what we call UML Profile+. Basically, in this step, an Ecore-based metamodel is generated that allows an instantiation of the UML stereotypes in the same way as any other UML element is instantiated.

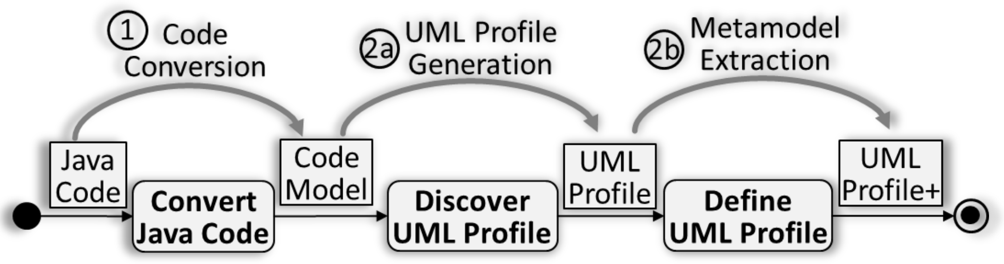


Figure 10 Conceptual Overview of the UML Profile Discoverer

3.1.7.3 Technical Specifications

- Languages
  - ATL for the model transformations
  - UML for the Profile representation
  - Java Profile for Java specifics
- Status: Alpha
- Release: 0.1

3.1.7.4 *Download and installation instructions*

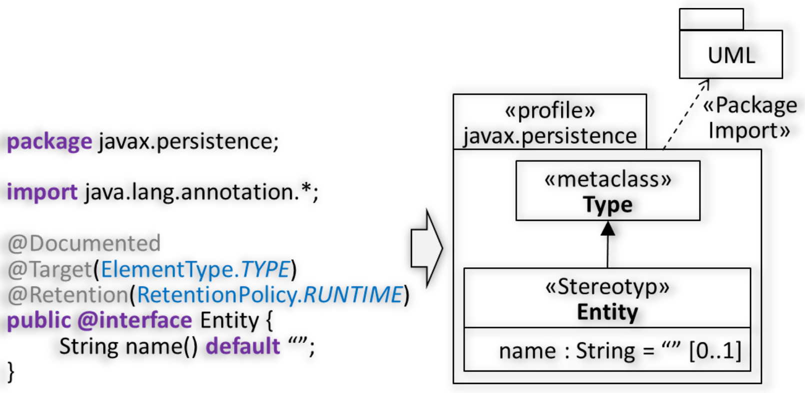
This component requires the following software to be installed:

- Eclipse Platform (get the latest Eclipse Modeling bundle)
- Eclipse MoDisco SDK
- Eclipse ATL SDK
- Eclipse UML2
- Java Profile

As for all components, the sources can be downloaded from the public repository (cf. section 2.5). Additionally, they can also be obtained directly on the component's Eclipse Labs page from the following location: https://code.google.com/a/eclipselabs.org/p/uml-profile-store/.

3.1.7.5 *User Manual*

The prototype is realized in terms of Eclipse plug-ins. They cover the implemented model transformation chain. A jUnit test case demonstrates the execution of the model transformation chain by relying on the persistence layer of the Java Petstore. In Figure 11, an excerpt of the JPA is depicted.



```
package javax.persistence;

import java.lang.annotation.*;

@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Entity {
        String name() default "";
}
```

Figure 11 Example-based Java Petstore persistence layer

When executing the *UML Profile discoverer*, the *Entity* annotation type declaration is transformed into the corresponding *Entity* stereotype. To ensure that the *Entity* stereotype provides at least similar capabilities as the corresponding Java declaration, the extension relationship references the UML class Type from the UML metamodel.

For accessing to more user information, please go to the component's Eclipse Labs page as indicated in the previous subsection.

3.1.7.6 *Licensing information*

This component is provided as open source under the Eclipse Public License [5].

3.2 .NET Support

3.2.1 Component "Generic UML Class Diagram Discoverer (structural part of the code) from C# source code or assembly – direct"

3.2.1.1 Overview

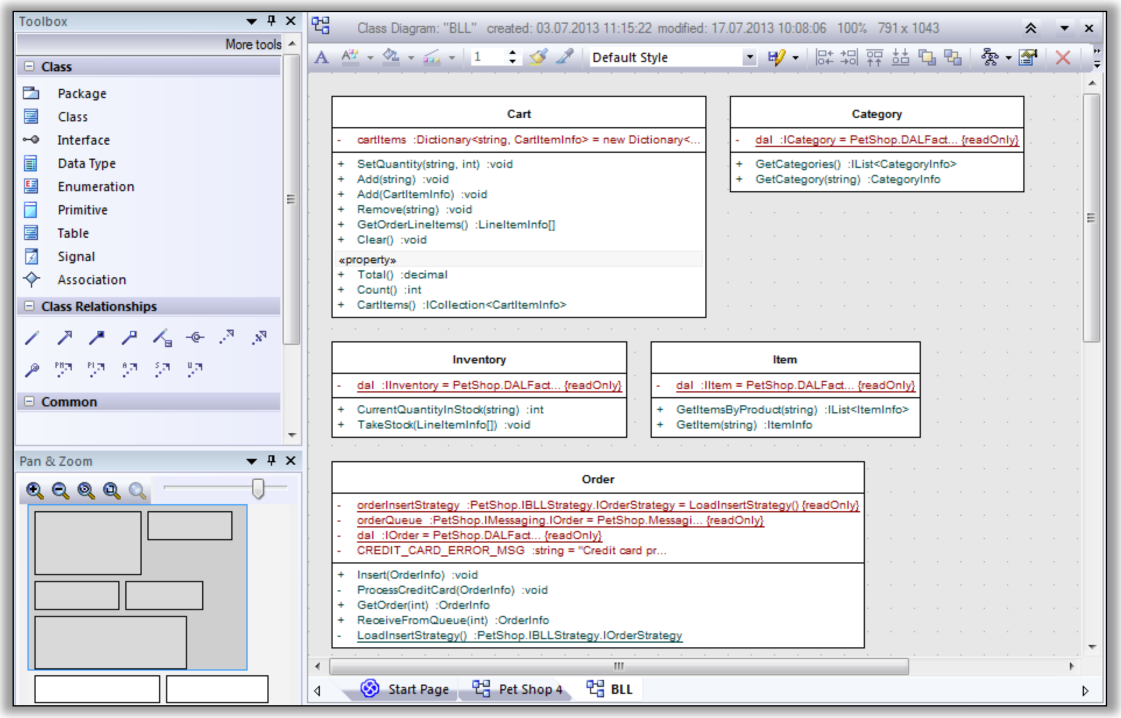
With this component built on top of Enterprise Architect and now part of its standard provided features, it is possible to reverse engineer the structural part of different programming languages, including notably C# as relevant in the context of ARTIST, and to create UML class diagrams for it.

This C# support comes with its own specific semantic constructs and not all of them can be mapped directly to UML Class Diagram. Hence, a mapping convention exists between C# and UML. In order to influence the way the UML class diagram is created from C#, it is possible to write/modify the grammar which is used to actually create the UML class diagram.

EA provides a framework for writing a grammar and importing it into the environment. An example language source file and an example Grammar for that language are provided in the *Code Samples* directory, which you can access from your installation directory (the default location is *C:\Program Files\Sparx Systems\EA*). Two other grammar files are also provided, illustrating specific aspects of developing Grammars. EA uses a variation of Backus–Naur Form (nBNF) to include processing instructions, the execution of which returns structured information from the parsed results in the form of an Abstract Syntax Tree (AST), which is used to generate the UML representation.

3.2.1.2 Functional description

In general it is possible to create a model from the source code contained in a folder structure or import a single source code file from the .Net framework.



Class Diagram: "BLL" created: 03.07.2013 11:15:22 modified: 17.07.2013 10:08:06 100% 791 x 1043

```
Cart
-  cartItems  :Dictionary<string, CartItemInfo> = new Dictionary<...
+  SetQuantity(string, int)  :void
+  Add(string)  :void
+  Add(CartItemInfo)  :void
+  Remove(string)  :void
+  GetOrderLineItems()  :LineItemInfo[]
+  Clear()  :void
«property»
+  Total()  :decimal
+  Count()  :int
+  CartItems()  :ICollection<CartItemInfo>

Category
-  dal  :ICategory = PetShop.DALFact... {readOnly}
+  GetCategories()  :IList<CategoryInfo>
+  GetCategory(string)  :CategoryInfo

Inventory
-  dal  :IInventory = PetShop.DALFact... {readOnly}
+  CurrentQuantityInStock(string)  :int
+  TakeStock(LineItemInfo[])  :void

Item
-  dal  :IItem = PetShop.DALFact... {readOnly}
+  GetItemsByProduct(string)  :IList<ItemInfo>
+  GetItem(string)  :ItemInfo

Order
-  orderInsertStrategy  :PetShop.IBLLStrategy.IOrderStrategy = LoadInsertStrategy() {readOnly}
-  orderQueue  :PetShop.IMessaging.IOrder = PetShop.Messagi... {readOnly}
-  dal  :IOrder = PetShop.DALFact... {readOnly}
-  CREDIT_CARD_ERROR_MSG  :string = "Credit card pr...
+  Insert(OrderInfo)  :void
-  ProcessCreditCard(OrderInfo)  :void
+  GetOrder(int)  :OrderInfo
+  ReceiveFromQueue(int)  :OrderInfo
-  LoadInsertStrategy()  :PetShop.IBLLStrategy.IOrderStrategy
```

Figure 12 UML Class Diagram model opened in Enterprise Architect

In all cases, the code is parsed based on a predefined grammar into an Abstract Syntax Tree (AST). Based on the established mapping conventions between UML Class Diagram and C#, a UML model is generated from the AST.

In case the same code has been already imported, the existing UML Class Diagram is either synchronized or overwritten depending on the case (this is a configurable option). Note that the models originating from Enterprise Architect can also be exchanged with other environments including the Eclipse/EMF one (cf. Java Support as presented in section 3.1).

3.2.1.3 Technical specifications

Enterprise Architect is a Windows 32 client application and uses a database as backend. The mainly used backend is a file base MSJet 3.5 or 4.0 Database. For bigger projects and multiple users (team environment) it is recommended to use a DBMS (e.g., MS SQL Server, Oracle, DB2 or MySql) instead of the file based MSJet database. In order to work in a team environment, it is possible to use a central DB which is shared by all team members or a version control system, which contains the models in a serialized form (XMI).

SVN, CVS, TFS and all SCC compliant version control systems are supported for the obtained models. To use version control, you activate it by default at package level. Nevertheless it is still possible to support finer-grain merge (e.g., at element level).

3.2.1.4 Download and installation instructions

This component is contained within the standard installation of Enterprise Architect. EA is a Windows application, which can be installed directly on the client's machine. For the installation process admin permissions are necessary. There is a single install file available. Before installing Enterprise Architect, the installer checks for an older version and uninstalls it automatically. Results of older versions can be used with any newer version. In few cases the model repository needs to be updated. In this case you are warned before the update process starts. Updated model repositories cannot be opened any more with the old version of EA. Thus, it is important to update all installed Enterprise Architect versions.

The download can be obtained directly from [13]. There is a fully functional 30-days free trial version that is made available, a valid license (as well as a valid user/key) is then needed to get the full version (cf. section 2.6 for more licensing information). Note that the sources are also available as mentioned in section 2.5 (currently for project consortium only).

3.2.1.5 User Manual

Import of Source Code from a Source Code Folder

Within an Enterprise Architect model the user has to select a package. In the context menu of the package select "Code Engineering | Import Source Directory …". Within the opened window select the source code type (here C#) and set a few further settings such as recursively parsing subfolders of the selected source code folder, creating diagrams for the generated UML class models, synchronizing existing classes, or deleting UML classes not found in the source code.

Import of a Single Source Code File

Within the Enterprise Architect's project browser a package should be selected. In the context menu of the package select "Code Engineering | Import from source file(s)", select the corresponding source code type (here C#) and specify the settings (see above).

3.2.1.6 Licensing information

This component is provided as open source but requires Sparx Enterprise Architect (cf. general licensing information in Section 2.6).

3.2.2 Component "Generic UML Sequence Diagram Discoverer (behavioural part of the code) from C# execution – direct"

3.2.2.1 Overview

This new component is based on the Visual Execution Analyzer (and is now part of the Sparx Enterprise Architect standard provided features,), which processes the structures and operations of the Sparx EA Model Driven Development Environment (MDDE). The MDDE provides tools to design, build and debug an application, all of them forming part of the Debugger facilities of Enterprise Architect.

One of the primary objectives of the Visual Execution Analysis feature is to enable working with the stack traces captured when debugging an application, in order to generate a UML Sequence Diagram. This is a practical way to have a good representation of what a C# program is doing during its execution phase.

3.2.2.2 Functional description

Enterprise Architect will record arguments to functions, and can optionally capture state transitions for a given State Machine. Calls to functions are aggregated on the class by default, but lifelines can also be created for each instance of an object.

This information can be integrated with existing system knowledge and test data to optimize code execution, reduce errors and determine why application failure and system crashes occur. If an application crashes, data corruption such as a stack overflow can prevent from diagnosing and rectifying the problem.

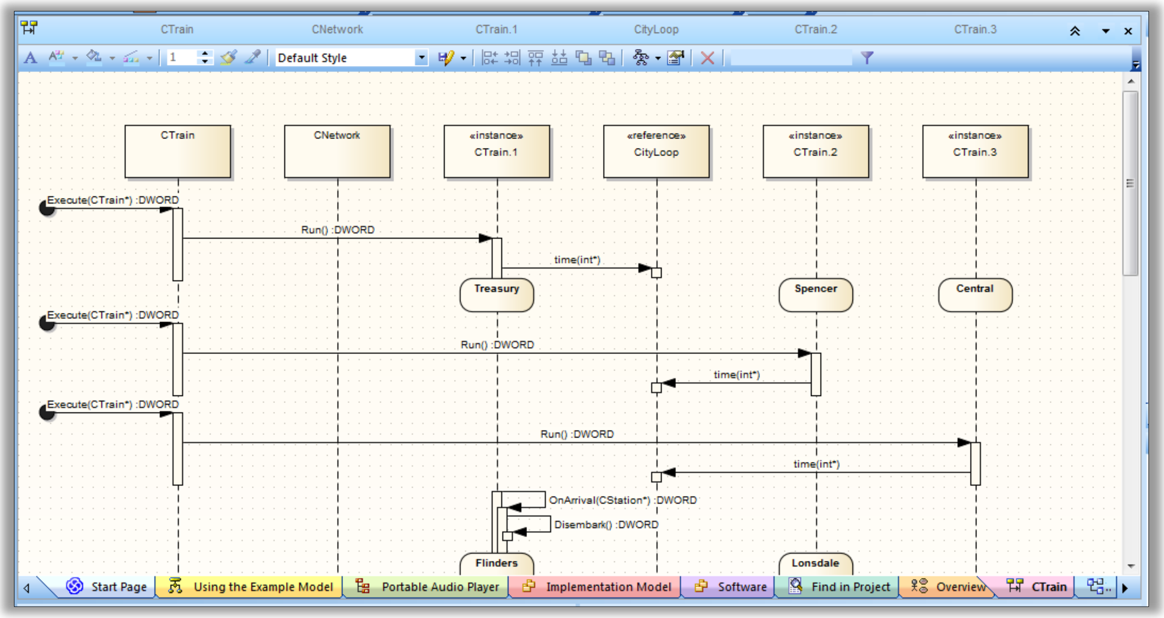


Figure 13 UML Sequence Diagram model opened in Enterprise Architect

However, the present component enables you to record a given execution sequence, as a UML Sequence Diagram, to provide a reliable source of information that might further explain why a crash occurred.

Such a UML Sequence Diagram can convey more detail and provide greater understanding than reading unfamiliar code that might have been written by someone else. It also makes it easier to document existing code when the model illustrates functions that are being called and the specific sequence of events that occur to produce a particular type of system behaviour.

Note that the models originating from Enterprise Architect can also be exchanged with other environments including the Eclipse/EMF one (cf. Java Support as presented in section 3.1).

3.2.2.3 Technical specifications

To avoid too complex sequence diagrams, it is possible to restrict the scope of the recorder by:

- Filter – specify templates in the Analyzer Script to exclude methods being recorded. For instance, *::get** will exclude all operations which start with “get”.
- Control Stack Depth - when recording particularly high-level points in an application, the stack frame count can result in a lot of information being collected; to achieve a quicker and clearer picture, it is better to limit the stack depth.

3.2.2.4 Download and installation instructions

The download and installation instructions of this present component are similar to the ones of the previously introduced Generic UML Class Diagram Discoverer from C# source code (cf. section 3.2.1.4).

3.2.2.5 User Manual

Before using the MDDE, check if you are using the correct edition: Enterprise Architect Professional, Corporate or extended editions.

Relevant source code files need to be linked to UML classes in the model (sufficient to be editable from the internal code editors) either by using specific MDA Transforms to generate stub code files or by using the previously presented Generic UML Class Diagram Discoverer from C# source code (cf. section 3.2.1).

Required external frameworks or compilers must be installed and operable. An Analyzer Script pointing to the external framework (at least to the compiler and debugger) needs to be configured to link to the external framework.

The Visual Execution Profiler enables you to quickly report on the most frequently called functions in a running application Tasks in an application that are taking more time than expected which functions are taking the most time in an application.

The Profiler only works with MS Native Windows applications, but can be used under WINE (Linux and Mac) to debug standard Windows applications deployed in a WINE environment.

The Profiler can generate a report that shows how these functions are called in relation to the application.

Select the desired class and press F12 to open the source code to debug, in the integrated source code editor.

Find the appropriate code line and right click in the left (Breakpoint) margin to bring up the breakpoint/marker context menu; select the required marker type: If the marker is denoted as a recording start point, the debugger immediately begins to trace all executed calls from that point for the breaking thread. Recording is stopped again when either the thread that is being captured terminates or the thread encounters a recording end point.

Select the appropriate package in the Project Browser, in which to store the Sequence diagram.

To create the UML Sequence Diagram model from all recorded sequences, right-click on the body of the window and select the Generate Sequence Diagram context menu option

3.2.2.6 Licensing information

This component is provided as open source but requires Sparx Enterprise Architect (cf. general licensing information in Section 2.6).

3.2.3 Component "Generic UML Class Diagram Discoverer from Microsoft library"

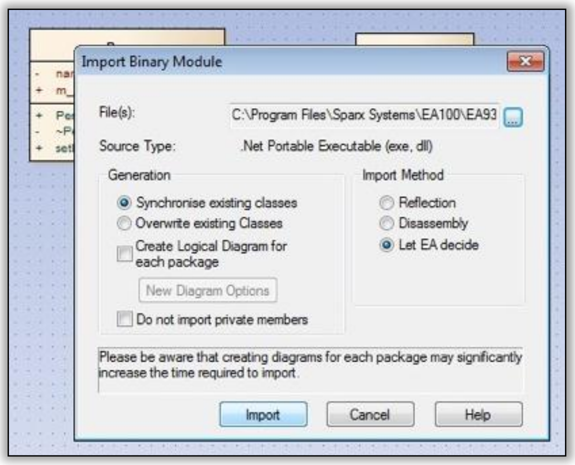
3.2.3.1 Overview

With this component built on top of Enterprise Architect and now part of its standard provided features, it is possible to reverse engineer the structural part of Microsoft libraries (i.e., DLL files), including notably C# ones as relevant in the context of ARTIST, and to create UML class diagrams for it.

Similarly to the Generic UML Class Diagram Discoverer from C# source code (cf. section 3.2.1), this library/DLL support comes with its own specific semantic constructs and not all of them can be mapped directly to UML Class Diagram. Thus, it is also possible to adapt (if needed) the mapping actually used to create the UML class diagram (cf. section 3.2.1.1).

3.2.3.2 Functional description

In general it is possible to create a model from an import of a binary module from the .Net or Java framework. In this case, the binary is parsed based on two different methods: either using reflection as provided by the .NET framework or by performing a disassembly of the considered library/DLL (as shown on next Figure 14).



Import Binary Module

File(s): C:\Program Files\Sparx Systems\EA100\EA93

Source Type: .Net Portable Executable (exe, dll)

Generation
- [✓ Synchronise existing classes]
- Overwrite existing Classes
- ☐ Create Logical Diagram for each package
- ☐ Do not import private members

Import Method
- Reflection
- Disassembly
- [✓ Let EA decide]

Please be aware that creating diagrams for each package may significantly increase the time required to import.

Figure 14 Options for discovering a UML Class Diagram model from a Microsoft library

In case the same binary has been already imported, the existing UML Class Diagram is either synchronized or overwritten depending on the case (this is a configurable option). Note that the models originating from Enterprise Architect can also be exchanged with other environments including the Eclipse/EMF one (cf. Java Support as presented in section 3.1).

3.2.3.3 Technical specifications

The technical specifications and requirements of this present component are similar to the ones of the previously introduced Generic UML Class Diagram Discoverer from C# source code (cf. section 3.2.1.3).

3.2.3.4 Download and installation instructions

The download and installation instructions of this present component are similar to the ones of the previously introduced Generic UML Class Diagram Discoverer from C# source code (cf. section 3.2.1.4).

3.2.3.5 User Manual

Within Enterprise Architects project browser a package should be selected. In the context menu of the package select "Code Engineering | Import Binary Module …", select the corresponding file and specify the necessary settings such as Reflection, Disassembly or Let EA decide. Make sure to specify in case of .NET the path to the correct PEVerify.exe, to make a proper reverse engineering possible.

This component provides facilities to model, develop, debug, profile and manage an application within the modeling environment. Outputs generated by the Visual Execution Analyzer supports the development process by:

- Giving you a better understanding of how your system works.
- Enabling to document system features automatically.
- Providing information on the sequence of events that lead to erroneous events or unexpected system behavior.

3.2.3.6 Licensing information

This component is provided as open source but requires Sparx Enterprise Architect (cf. general licensing information in Section 2.6).

3.2.4 Component "SharePoint Data Model Discoverer from SharePoint database"

3.2.4.1 Overview

Customizations or extensions in Microsoft SharePoint [11] are typically installed by means of so-called Features. A feature could be used to define custom data structures such as Site Columns, Content Types, List Instances and so on. The migration of a SharePoint-based solution (using these extensions) towards a SaaS solution (independent of SharePoint) requires us to extract these features that are persisted in the SharePoint database and move them towards a stand-alone database. The purpose of this component is to provide such a transformation chain for the List Instance data structure.

The goal for this component is to implement the *Part 1 of a 3-phase transformation chain*, i.e., produce input for another phase in which the SP data model is going to be transformed to a Database Model (phase 2, Model Understanding), independently from a targeted platform, for it to be eventually forward engineered to an actual Database Schema (phase 3, Model Understanding).

3.2.4.2 Functional description

The component provides a transformation pipeline (together with the supporting metamodels and transformations) involving a number of steps to be able to convert a SharePoint feature (particularly a List Instance) towards a SharePoint Data Model. This is achieved by means of a number of steps (see figure below).

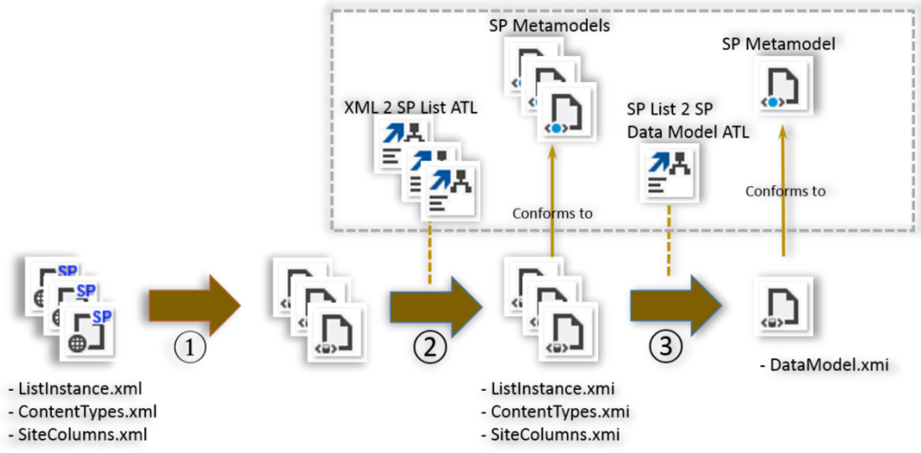


Figure 15 SharePoint Data Model Discoverer: Architecture Overview

In step (1), XML models are discovered from XML files conforming to the SharePoint schema definitions [11]. This is done using the Generic XML Model Discoverer as introduced before. Here, three files are being produced which are direct mappings of the original input files. In step (2), the output of step (1) is being transformed, each via a dedicated ATL transformation, to intermediate models, each conforming to a dedicated metamodel. For this, a metamodel for the List Instance, Content Type and Site Column have been provided. Finally, in step (3), a combined view is given via a single transformation of multiple models towards one single model, conforming to a dedicated metamodel.

3.2.4.3 Technical specifications

As a summary, here are the concrete technologies that have been used and deployed in the context of this component:

- XML metamodel and XML model discovery – Eclipse MoDisco [4]
- ATL transformations – Eclipse ATL [1]

General information:

- Status: Alpha
- Release: 0.1

3.2.4.4 Download and installation instructions

This component requires the following software to be installed:

- Eclipse Platform (get the latest Eclipse Modeling bundle)
- Eclipse MoDisco SDK
- Eclipse ATL SDK

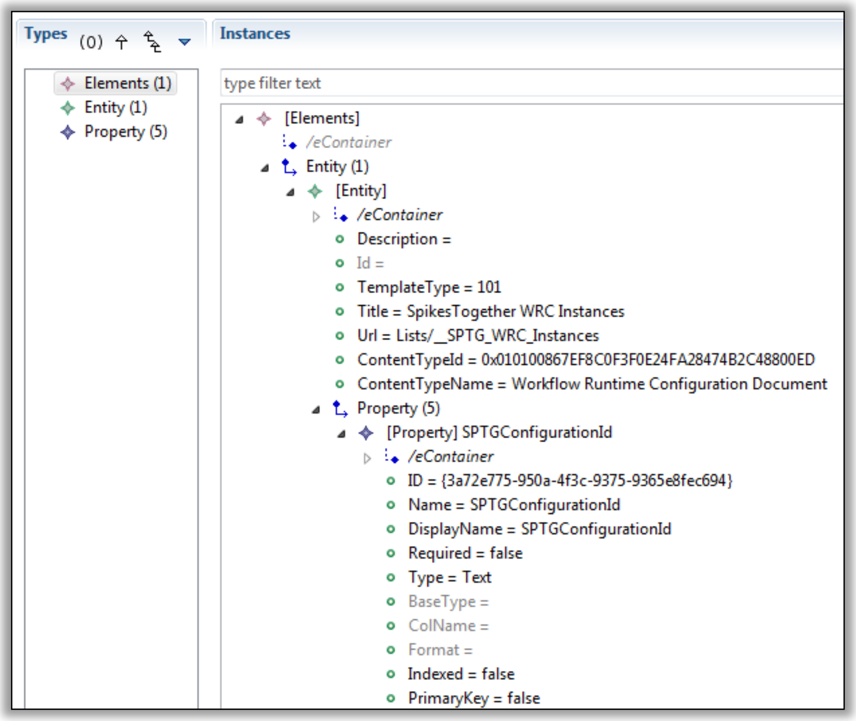
As for the other newly developed components, the sources can be downloaded from the public repository (cf. section 2.5).

3.2.4.5 User Manual

In order to setup the transformation pipeline, it would be necessary to create a new ATL project (File → New → ATL Project). The files required can now be imported into your project within Eclipse (File → Import…). Select the different original input files and also import them into your project.

The first step of the transformation chain can be performed by executing the XML discoverer of MoDisco (right-click an XML file Discovery → Discoverers → Discover XML Model). This has to be done for each of the three input files used.

Step 2 of the transformation chain can be performed by running an ATL transformation. Firstly, the three XMI files produced in step 1 need to be transformed to models conforming to their own metamodel. This can be done by defining a new Run Configuration (Run → Run Configurations) and creating an ATL transformation configuration. The XML metamodel together with one of *ListInstance.ecore*, *ContentType.ecore*, *SiteColumn.ecore* metamodels should be set as the metamodels. Next one of the XML models should be set as source model and a new XMI should be set as target model. Step 3 involves a similar approach. A new ATL transformation configuration has to be created, specifying the three *ListInstance.ecore*, *ContentType.ecore*, *SiteColumn.ecore* metamodels together with the *SPDataModel.ecore* metamodel.



```
Elements (1)
Entity (1)
Property (5)
```

```
[Elements]
  /eContainer
  Entity (1)
    [Entity]
      /eContainer
      Description =
      Id =
      TemplateType = 101
      Title = SpikesTogether WRC Instances
      Url = Lists/__SPTG_WRC_Instances
      ContentTypeId = 0x010100867EF8C0F3F0E24FA28474B2C48800ED
      ContentTypeName = Workflow Runtime Configuration Document
      Property (5)
        [Property] SPTGConfigurationId
          /eContainer
          ID = {3a72e775-950a-4f3c-9375-9365e8fec694}
          Name = SPTGConfigurationId
          DisplayName = SPTGConfigurationId
          Required = false
          Type = Text
          BaseType =
          ColName =
          Format =
          Indexed = false
          PrimaryKey = false
```

Figure 16 SharePoint Data model opened in the MoDisco Model Browser

Give the output of step to as source models and a new SharePoint Data model should be set to be generated as target model (as shown on Figure 16).

3.2.4.6 Licensing information

This component is provided as open source under the Eclipse Public License [5].

3.3 Generic Support

3.3.1 Component “Generic XML Model Discover and Aggregator from XML documents”

3.3.1.1 *Overview*

This component offers both a generic metamodel (based on the XML Language Specification) and the Generic XML Model Discoverer. This latter allows obtaining complete XML models out of any XML document. The XML model represents all the content of the initial XML document, independently from the corresponding XML Schema or DTD.

The Generic XML Model Discoverer is provided by the Eclipse MoDisco project (initiated and created within the context of the IST-FP6 MODELPLEX EU project). However, it is practically reused in ARTIST (as an individual component or to provide inputs to some newly developed components) and so is considered as a base component from the ARTIST Model Discovery Toolbox.

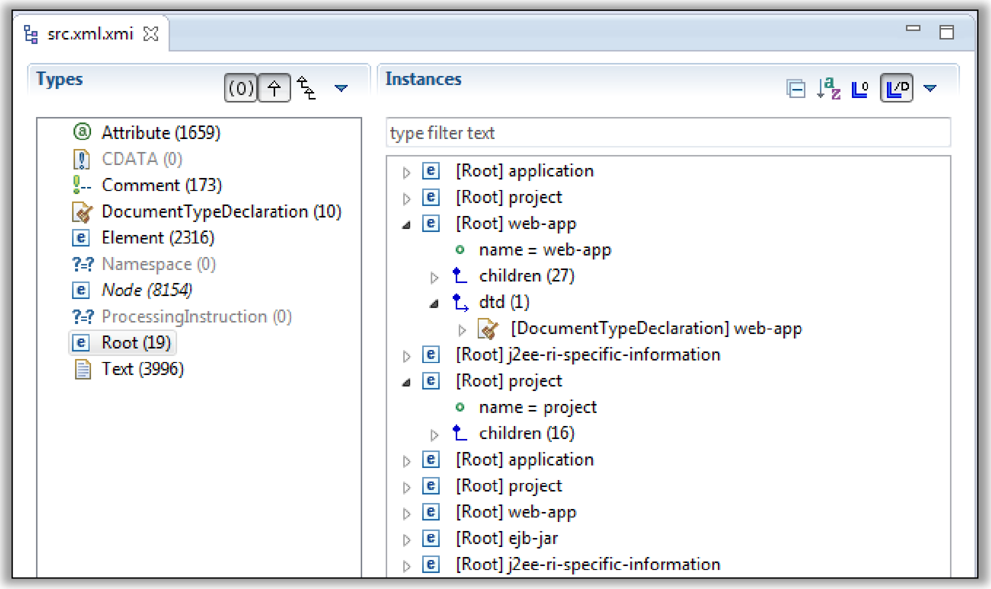
The Aggregator part has been newly developed in the context of ARTIST, following the needs of some of the partners in the project, in order to gather in one global XML model the information from several different XML (configuration) files that are actually related together.

For more information, reference documentation or download/installation instructions about the Generic XML Model Discoverer, please go to the MoDisco homepage [4]. For details on the Aggregator additional part, please take a look to the next subsections.

3.3.1.2 *Functional description*

This component currently considers:

- As input: several XML documents (files) that are grouped in a same project or directory for instance.
- As output: a complete XML model containing data extracted out of each one of the XML documents taken as inputs (cf. Figure 17).



```
Attribute (1659)
CDATA (0)
Comment (173)
DocumentTypeDeclaration (10)
Element (2316)
Namespace (0)
Node (8154)
ProcessingInstruction (0)
Root (19)
Text (3996)
```

```
[Root] application
[Root] project
[Root] web-app
    name = web-app
    children (27)
    dtd (1)
        [DocumentTypeDeclaration] web-app
[Root] j2ee-ri-specific-information
[Root] project
    name = project
    children (16)
[Root] application
[Root] project
[Root] web-app
[Root] ejb-jar
[Root] j2ee-ri-specific-information
```

Figure 17 XML (Aggregated) model opened in the MoDisco Model Browser

3.3.1.3 Technical specifications

The Aggregator has been embedded in and packaged as a MoDisco Discoverer (currently named *fr.inria.artist.modisco.xml.launcher*).

When executed, the aggregator component actually starts from a project or folder containing several XML documents (files) and then:

- Searches for XML files in the folder passed as input
- Once an XML document has been found, it launches the Generic XML Model Discoverer on it. The obtained XML model is not serialized at this time (and just kept in memory).
- The root element of the obtained XML model is stored in a central XML model shared by the whole discovery process.

At the end of the process, when all the XML files have been processed iteratively, the obtained XML model is finally serialized at the same level as the directory.

As a summary, Eclipse MoDisco [4] has been used and deployed in the context of this component for the XML metamodel and XML generic model discovery support.

3.3.1.4 Download and installation instructions

Thus, this component requires the following software to be installed:

- Eclipse Platform (get the latest Eclipse Modeling bundle)
- Eclipse MoDisco SDK

As for the other newly developed components, the sources can be downloaded from the public repository (cf. section 2.5).

3.3.1.5 User manual

To use the component, you can start from a folder containing one or several XML files (even nested in sub-folders) and then:

- Right-click on this folder.
- Select the following menus *"Discovery / Discoverers / Multiple XML file discoverer".*
- The same kind of window as the one shown in Figure 6 will open and ask whether you want to serialize the output XML model and set manually its location.
- Validate your choices in the window, and run the discovery process. The resulting XML model will by default (i.e., if not specified differently) be saved at the level of the initially selected folder.

The obtained XML model can finally be navigated using the MoDisco Model Browser for instance (cf. Figure 17 for seeing an example).

3.3.1.6 Licensing information

This component is provided as open source under the Eclipse Public License [5].

4 Conclusion and Future Work

The present deliverable is about introducing the initial version of the ARTIST Model Discovery Toolbox. The main objective of this toolbox is to provide the migration engineers with the needed set of components so that they can perform the very first step of their MDRE processes. Taking as inputs the legacy artifacts composing the legacy systems to be migrated (potentially coming from the Java EE or .NET world), the toolbox allows producing as outputs the base models that are sufficient and relevant for continuing the processes (i.e., dealing with Model Understanding).

During this first year of the ARTIST project, the foundations have been set in order to build this initial version of the ARTIST Model Discovery Toolbox. As a summary, the work performed until now has mainly consisted in:

1. Identifying the already available support that is relevant and useful in the context of ARTIST and its use cases. Thus, two main platforms have been chosen in order to elaborate further on the two main families of support (basically Eclipse MoDisco for Java EE and Sparx EA for .NET).
2. Selecting the interface or exchange format with the following Model Understanding step (and corresponding toolbox, cf. D8.3.2). Generic UML models and/or more low-level models (e.g., KDM, Java, C#) are going to be shared from this Model Discovery toolbox to the Model Understanding one.
3. Prototyping new discovery components according to the missing support or to the requirements expressed by use case providers or Model Understanding actors. For instance, new generic UML discoverers have been/are being developed.

The current version of the toolbox is mainly resulting from the various experiments conducted during the first year of the project on both the commonly agreed PetStore example and some particular project's use cases. As a consequence, most of the newly developed components, relying on previously existing components or built from scratch, are still work in progress. Thus, the objective for the second year of the project (and for the delivery of the second version of the ARTIST Model Discovery Toolbox) is twofold:

- Consolidate the components prototyped during the first year and already provided as part of the initial version of the toolbox.
- Work on new components, when relevant and possible according to the available resources, corresponding to concrete deployments/uses of the toolbox on project's use cases. For example, based on some feedback we already foresee the need for:
  - A C# metamodel and corresponding base (partial) discoverer (.NET-specific support).
  - A more advanced support for the discovery of the behavioural aspects of the source code (both for Java and C#).
  - Potentially other components to be determined during the running of the project.

5 Reference

[1] Eclipse ATL tool. Available at: http://www.eclipse.org/atl/
[2] Eclipse EMF tool. Available at: http://www.eclipse.org/modeling/emf/
[3] Eclipse JDT tool. Available at: http://www.eclipse.org/jdt/
[4] Eclipse MoDisco tool. Available at: http://www.eclipse.org/MoDisco/
[5] Eclipse Public License terms. Available at: http://www.eclipse.org/legal/epl-v10.html
[6] Eclipse UML2 metamodel. Available at: http://www.eclipse.org/uml2/
[7] OMG ADM Task Force. Homepage: http://adm.omg.org/
[8] OMG fUML specification. Available at: http://www.omg.org/spec/FUML/
[9] OMG KDM specification. Available at: http://www.omg.org/technology/kdm/
[10] OMG UML specification. Available at: http://www.uml.org/
[11] SharePoint schema definition. Available at: http://msdn.microsoft.com/en-us/library/ms476062.aspx
[12] Sparx Enterprise Architect toolset. Available at: http://www.sparxsystems.com/products/ea/
[13] Sparx Enterprise Architect (trial version) download. Available at: http://www.sparxsystems.com/products/ea/trial.html