**ARTIST**

**FP7-317859**



***Advanced software-based seRvice provisioning and migraTIon of legacy Software***

**Deliverable D8.2.2**

**Components for Model Discovery from Legacy Technologies**

|  |  |
| --- | --- |
| **Editor(s):** | Hugo Bruneliere, Javier Canovas |
| **Responsible Partner:** | Inria |
| **Status-Version:** | Draft – v0.1 |
| **Date:** | 23/07/2014 |
| **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:** | 30/09/2014 |

|  |  |
| --- | --- |
| **Workpackage responsible for the Deliverable:** | WP8 – Legacy Product Analysis by Reverse Engineering |
| **Editor(s):** | Inria |
| **Contributor(s):** | Inria, ATOS, TUWien, Sparx, Spikes, TECNALIA |
| **Reviewer(s):** | XXX (YYY) |
| **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# existing source artefacts, 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 generally licensed under the Eclipse Public License (EPL, open source) for the Java EE support, it is indicated otherwise (notably for the .NET support). |

**Document Description**

**Document Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| *Version* | *Date* | *Modifications**Introduced* | |
| *Modification Reason* | *Modified by* |
| v0.1 | 23/07/14 | First draft version (starting from already submitted D8.2.1)  Integration of several updates and additions (e.g. “Reusability Trace Tool”) all along the document. | Hugo Bruneliere (Inria), Javier Canovas (Inria)  Contributions: Matthieu Allon (Inria), Konrad Wieland (Sparx), Alexander Bergmayr (TUWien), Bram Pellens (Spikes), Gorka Echevarria Velez (TECNALIA) |
| v0.2 |  |  |  |
| v0.3 |  |  |  |
|  |  |  |  |
| V1.0 | XX/09/14 | Complete version for submission to EC (integrating comments from internal review) | Hugo Bruneliere (Inria) |

# Table of Contents

[Table of Contents 4](#_Toc397350219)

[Table of Figures 8](#_Toc397350220)

[Table of Tables 8](#_Toc397350221)

[Terms and abbreviations 9](#_Toc397350222)

[Executive Summary 10](#_Toc397350223)

[1 Introduction 12](#_Toc397350224)

[1.1 About this deliverable 12](#_Toc397350225)

[1.2 Document structure 12](#_Toc397350226)

[2 ARTIST Model Discovery Toolbox: Overview 13](#_Toc397350227)

[2.1 Functional description 13](#_Toc397350228)

[2.2 Fitting into overall ARTIST solution 14](#_Toc397350229)

[2.3 Technical description 15](#_Toc397350230)

[2.4 Toolbox global architecture 17](#_Toc397350231)

[2.5 General Delivery Information 17](#_Toc397350232)

[2.6 General Licensing Information 18](#_Toc397350233)

[3 ARTIST Model Discovery Toolbox: Components Details 19](#_Toc397350234)

[3.1 Java EE Support 19](#_Toc397350235)

[3.1.1 Component “Java Code Model Discoverer from Java source code” 19](#_Toc397350236)

[3.1.2 Component “KDM Code Model Discoverer from Java source code” 20](#_Toc397350237)

[3.1.3 Component “Generic UML Class Diagram Discoverer (structural part of the code) from Java source code – via KDM” 21](#_Toc397350238)

[3.1.4 Component “Generic UML Class Diagram Discoverer (structural part of the code) from Java source code – direct” 22](#_Toc397350239)

[3.1.4.1 Overview 22](#_Toc397350240)

[3.1.4.2 Functional description 22](#_Toc397350241)

[3.1.4.3 Technical specifications 23](#_Toc397350242)

[3.1.4.4 Download and installation instructions 23](#_Toc397350243)

[3.1.4.5 User manual 23](#_Toc397350244)

[3.1.4.6 Licensing information 24](#_Toc397350245)

[3.1.5 Component “Generic UML Activity Diagram Discoverer (behavioural part of the code) from Java source code – direct” 24](#_Toc397350246)

[3.1.5.1 Overview 24](#_Toc397350247)

[3.1.5.2 Functional description 24](#_Toc397350248)

[3.1.5.3 Technical specifications 25](#_Toc397350249)

[3.1.5.4 Download and installation instructions 25](#_Toc397350250)

[3.1.5.5 User manual 26](#_Toc397350251)

[3.1.5.6 Licensing information 26](#_Toc397350252)

[3.1.6 Component “KDM Source Model Discoverer from Java project” 26](#_Toc397350253)

[3.1.7 Component “Generic UML Profile Discoverer from Java API/library” 27](#_Toc397350254)

[3.1.7.1 Overview 27](#_Toc397350255)

[3.1.7.2 Functional Description 28](#_Toc397350256)

[3.1.7.3 Technical Specifications 29](#_Toc397350257)

[3.1.7.4 Download and installation instructions 29](#_Toc397350258)

[3.1.7.5 User Manual 29](#_Toc397350259)

[3.1.7.6 Licensing information 30](#_Toc397350260)

[3.2 SQL Support 31](#_Toc397350261)

[3.2.1 Component “SQL Model Discoverer from SQL instructions” 31](#_Toc397350262)

[3.2.1.1 Overview 31](#_Toc397350263)

[3.2.1.2 Functional description 31](#_Toc397350264)

[3.2.1.3 Technical specifications 31](#_Toc397350265)

[3.2.1.4 Download and installation instructions 32](#_Toc397350266)

[3.2.1.5 User manual 32](#_Toc397350267)

[3.2.1.6 Licensing information 32](#_Toc397350268)

[3.2.2 Component “Generic UML Object Diagram Discoverer from SQL model” 32](#_Toc397350269)

[3.2.2.1 Overview 32](#_Toc397350270)

[3.2.2.2 Functional description 33](#_Toc397350271)

[3.2.2.3 Technical specifications 33](#_Toc397350272)

[3.2.2.4 Download and installation instructions 34](#_Toc397350273)

[3.2.2.5 User manual 34](#_Toc397350274)

[3.2.2.6 Licensing information 35](#_Toc397350275)

[3.3 .NET Support 35](#_Toc397350276)

[3.3.1 Component “Generic UML Class Diagram Discoverer (structural part of the code) from C# source code or assembly – direct” 35](#_Toc397350277)

[3.3.1.1 Overview 35](#_Toc397350278)

[3.3.1.2 Functional description 35](#_Toc397350279)

[3.3.1.3 Technical specifications 36](#_Toc397350280)

[3.3.1.4 Download and installation instructions 36](#_Toc397350281)

[3.3.1.5 User Manual 37](#_Toc397350282)

[3.3.1.6 Licensing information 37](#_Toc397350283)

[3.3.2 Component “Generic UML Sequence Diagram Discoverer (behavioural part of the code) from C# execution – direct” 37](#_Toc397350284)

[3.3.2.1 Overview 37](#_Toc397350285)

[3.3.2.2 Functional description 37](#_Toc397350286)

[3.3.2.3 Technical specifications 38](#_Toc397350287)

[3.3.2.4 Download and installation instructions 38](#_Toc397350288)

[3.3.2.5 User Manual 39](#_Toc397350289)

[3.3.2.6 Licensing information 39](#_Toc397350290)

[3.3.3 Component “Generic UML Class Diagram Discoverer from Microsoft library” 39](#_Toc397350291)

[3.3.3.1 Overview 39](#_Toc397350292)

[3.3.3.2 Functional description 40](#_Toc397350293)

[3.3.3.3 Technical specifications 40](#_Toc397350294)

[3.3.3.4 Download and installation instructions 40](#_Toc397350295)

[3.3.3.5 User Manual 40](#_Toc397350296)

[3.3.3.6 Licensing information 41](#_Toc397350297)

[3.3.4 Component “SharePoint Data Model Discoverer from SharePoint database” 41](#_Toc397350298)

[3.3.4.1 Overview 41](#_Toc397350299)

[3.3.4.2 Functional description 41](#_Toc397350300)

[3.3.4.3 Technical specifications 42](#_Toc397350301)

[3.3.4.4 Download and installation instructions 42](#_Toc397350302)

[3.3.4.5 User Manual 42](#_Toc397350303)

[3.3.4.6 Licensing information 43](#_Toc397350304)

[3.4 Generic Support 44](#_Toc397350305)

[3.4.1 Component “Generic XML Model Discover and Aggregator from XML documents” 44](#_Toc397350306)

[3.4.1.1 Overview 44](#_Toc397350307)

[3.4.1.2 Functional description 44](#_Toc397350308)

[3.4.1.3 Technical specifications 45](#_Toc397350309)

[3.4.1.4 Download and installation instructions 45](#_Toc397350310)

[3.4.1.5 User manual 45](#_Toc397350311)

[3.4.1.6 Licensing information 45](#_Toc397350312)

[3.5 Component “Reusability Trace Tool” 46](#_Toc397350313)

[3.5.1 Overview 46](#_Toc397350314)

[3.5.2 Functional description 46](#_Toc397350315)

[3.5.3 Technical specifications 47](#_Toc397350316)

[3.5.4 Download and installation instructions 47](#_Toc397350317)

[3.5.5 User manual 48](#_Toc397350318)

[3.5.6 Licensing information 50](#_Toc397350319)

[4 Conclusion and Future Work 50](#_Toc397350320)

[5 Reference 52](#_Toc397350321)

# Table of Figures

[Figure 1 Excerpt of the ARTIST Architecture, the “Model Discovery” component 15](#_Toc397350322)

[Figure 2 ARTIST Model Discovery Toolbox: Overall Architecture 17](#_Toc397350323)

[Figure 3 Java Code model opened in the MoDisco Model Browser 19](#_Toc397350324)

[Figure 4 KDM Code model opened in the MoDisco Model Browser 20](#_Toc397350325)

[Figure 5 UML Class Diagram model opened in the MoDisco Model Browser 21](#_Toc397350326)

[Figure 6 Parameter selection for the UML Class Diagram Model Discoverer 24](#_Toc397350327)

[Figure 7 UML Activity Diagram model opened in the MoDisco Model Browser 26](#_Toc397350328)

[Figure 8 KDM Source model opened in the MoDisco Model Browser 27](#_Toc397350329)

[Figure 9 UML Profile model discovered from a Java library (*jar* file) 28](#_Toc397350330)

[Figure 10 Conceptual Overview of the UML Profile Discoverer 29](#_Toc397350331)

[Figure 11 Example-based Java Petstore persistence layer 29](#_Toc397350332)

[Figure 12 SQL model opened in the MoDisco Model Browser 32](#_Toc397350333)

[Figure 13 UML Object Diagram model opened in the MoDisco Model Browser 34](#_Toc397350334)

[Figure 14 UML Class Diagram model opened in Sparx Enterprise Architect 36](#_Toc397350335)

[Figure 15 UML Sequence Diagram model opened in Sparx Enterprise Architect 38](#_Toc397350336)

[Figure 16 Options for discovering a UML Class Diagram model from a Microsoft library 40](#_Toc397350337)

[Figure 22 RTT activation in Eclipse 48](#_Toc397350338)

[Figure 23 RTT wizard 48](#_Toc397350339)

[Figure 24 RTT wizard for Presentation layer 48](#_Toc397350340)

[Figure 25 RTT wizard for Business layer 49](#_Toc397350341)

[Figure 26 RTT wizard for reusable code selection 50](#_Toc397350342)

# Table of Tables

[Table 1 Mapping (excerpt) used in the Java-to-UML Class diagram transformation 22](#_Toc397350348)

[Table 2 Mapping (excerpt) used in the Java-to-UML Activity diagrams transformation 25](#_Toc397350349)

[Table 3 Mapping used in the SQL model injection 31](#_Toc397350350)

[Table 4 Mapping used in the SQL\UML profiled-to-UML Object diagram transformation 33](#_Toc397350351)

# 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

Overall content of the deliverable

As ARTIST is dealing with the migration of already existing 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 already existing 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 existing system. Thus, a Model Driven Reverse Engineering (MDRE) tooling is needed in order to (semi-)automatically produce these models out of the software artefacts 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 existing 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.

Context of the deliverable, how it is related to the overall project and target project work packages

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 software artefacts of a given existing system. The models produced are base models in the sense that they are direct/systematic representations of the content of the software artefacts, 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. D8.3.2) is then in charge of performing these next operations.

List and summarize the main sections of the document and rationale for them

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 (targeting these two main technical spaces). In addition to offering a big picture of the toolbox itself (architecture, functional and technical features, positioning in ARTIST), this deliverable is also similarly giving more insights on each one of the individual components developed and/or proposed by the different partners involved in this particular aspect of the project. The document also addresses the support for traceability which is required throughout the ARTIST modernization process. Such a support is described as part of this deliverable and Task 8.2/WP8, even though it is obviously relevant and useful within the context of other WPs.

Highlight main results and innovations brought by the deliverable to the project

The main innovation of the proposed toolbox is that it is open and extensible. Thus, the base support currently offered relies on a set of well-established IDEs and related capabilities: the Eclipse Modeling project for the Java technology and the Sparx Enterprise Architect solution for the .NET technology. They have been/are being/will be reused and extended in order to design and build the new components implementing the required functionalities as progressively identified within the course of the project. In any case, exchanges between these two environments are ensured via model import/export capabilities using the UML standard as a base (and this is also true with other tasks and WPs such as T8.3 and WP9).

Mention future related work in the project (next versions content, other deliverables to be produced, etc)

This deliverable presents the current version of the ARTIST Model Discovery Toolbox at M24, resulting from the consolidation and extension of the initial version of this same toolbox as provided at M12 (cf. ARTIST D8.2.1). It will be followed at M30 by a last version (in the context of ARTIST at least) of this toolbox that is planned to go a step further by offering more stable versions of the discovery components developed all along the project, a more advanced support for the main technologies considered in the project (i.e., Java and C#) and for the reuse of traces during the ARTIST migration process. TO BE UPDATED AT M24!

# Introduction

Within the context of ARTIST or more generally in any other software modernization project, different kinds of software artefacts 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 (already existing) 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 artefacts 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 second version (over three all along the project) of the concrete tool support for this first Model Discovery phase. This Model Discovery support is conceptually detailed as part of *D8.2 “Methodologies and Techniques for Model Discovery”*. For your information, note that the corresponding Model Understanding support is described in *D8.3.2 “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 existing 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.

## 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.

## 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.

# ARTIST Model Discovery Toolbox: Overview

## Functional description

The purpose of this second version of the ARTIST Model Discovery Toolbox is to provide modernization engineers with an extended base support for performing the mandatory Model Discovery step of their MDRE processes. The overall idea is, given the different software artefacts composing the existing 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 existing 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 technology and Sparx Enterprise Architect for the .NET technology. They have been/are being/will be reused and extended in order to design and build the new components implementing the required functionalities.

In terms 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 the features developed during these two first years):

* From Java EE artefacts
  + 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 (complete)**
    - *KDM Source model (artefacts inventory)*
  + From Java APIs/libraries (currently in Jars)
    - **Generic UML profile (in progress)**
* From .NET artefacts
  + 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 artefacts
  + *Generic XML document model (complete)* + **XML model aggregator (complete)**
* Code-Model-Code Traceability support
  + **Reusable Traces (in progress)**

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 software 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 (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 software artefacts 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 existing 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 artefacts (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 software 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 version.

## 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 existing 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 software artefacts composing the systems to be modernized. The expected support for both the Java and .NET technologies is shown in Figure 1.

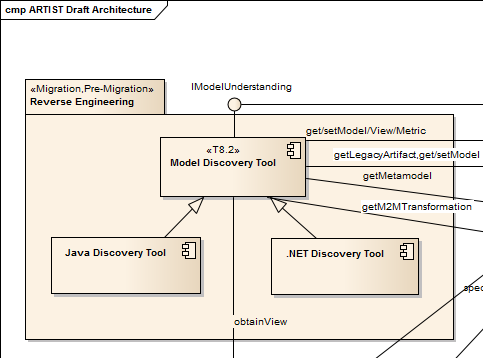


Figure 1 Excerpt of the ARTIST Architecture, the “Model Discovery” component

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.

## 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 technology;
* Sparx Enterprise Architect [13] for the .NET technology.

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 software artefacts 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 [11] is used as the base for models produced as outputs of the toolbox while the OMG KDM standard metamodel [10] is also considered as an alternative option for interoperability purposes. For the Java technology, 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 technology, the UML2 metamodel implementation is the one fitted with Sparx Enterprise Architecture [13]. 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 technologies.

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 the features developed during these two first years):

* From Java EE artefacts
  + 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 (complete)**
      * Newly developed version directly from the Java code model – implementation in ATL
    - *KDM Source model (artefacts 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 artefacts
  + 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 Architect extension – implementation in C#/.NET
  + From Microsoft libraries (currently DLL files)
    - **Generic UML class diagram (in progress)**
      * Newly developed Enterprise Architect 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 artefacts
  + *Generic XML document model (complete)* + **XML model aggregator (complete)**
    - MoDisco base support reused and extended in the project – implementation in Java
* Code-Model-Code Traceability support
  + **Reusability Trace Tool (in progress)**
    - Newly developed prototype – current implementation in Java for the Java technology

## 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.

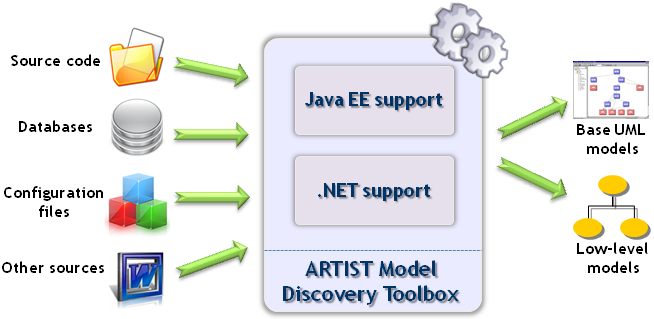


Figure 2 ARTIST Model Discovery Toolbox: Overall Architecture

The Model Discovery toolbox is taking as inputs different kinds of software artefacts from both the Java EE and .NET technologies, 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 technology, 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 technologies.

## General Delivery Information

The sources of the different components newly developed during these two first years of the ARTIST project have been grouped in a Zip file that has been put on the ARTIST (Livelink) repository at Deliverables/M12 folder for convenience to reviewers.

However, they are also generally available for ARTIST partners from <https://github.com/artist-project/ARTIST-Tooling/tree/master/migration/application-discovery-understanding/MDT>

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 [13]), 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).

## 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 technologies/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).

# 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 existing 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.

## Java EE Support

### 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).

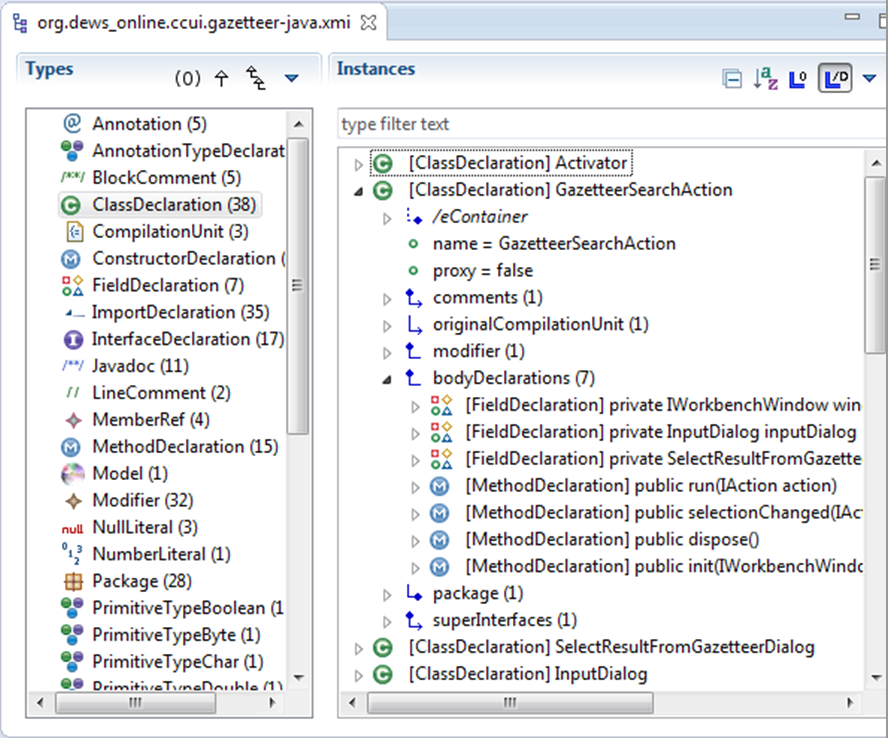


Figure 3 Java Code model opened in the MoDisco Model Browser

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].

### 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 [8].

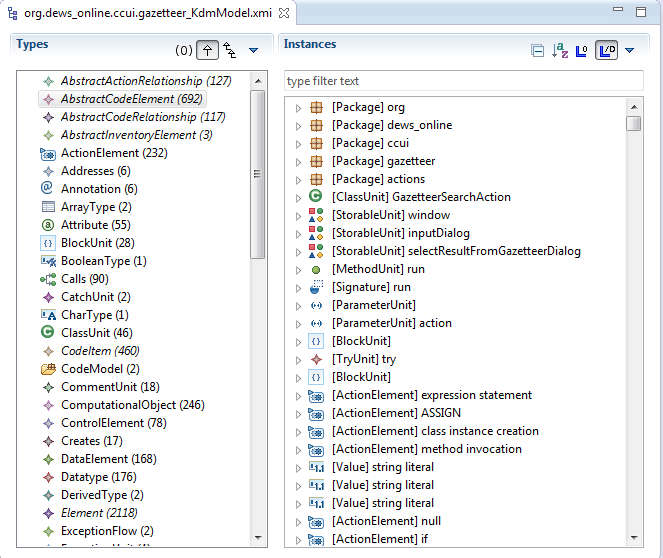


Figure 4 KDM Code model opened in the MoDisco Model Browser

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].

### 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).

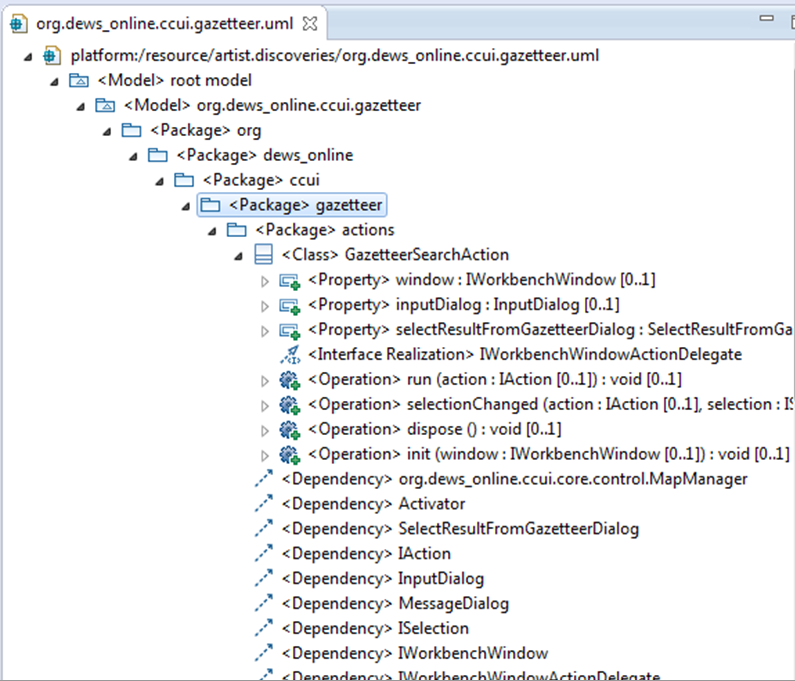


Figure 5 UML Class Diagram model opened in the MoDisco Model Browser

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].

### Component “Generic UML Class Diagram Discoverer (structural part of the code) from Java source code – direct”

#### 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 [11]. 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 actually covered by the Generic UML Activity Diagram Model Discoverer (cf. section 3.1.5).

#### 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 OR Class OR Enumeration] |
| 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.*

#### 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

#### 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).

#### 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.

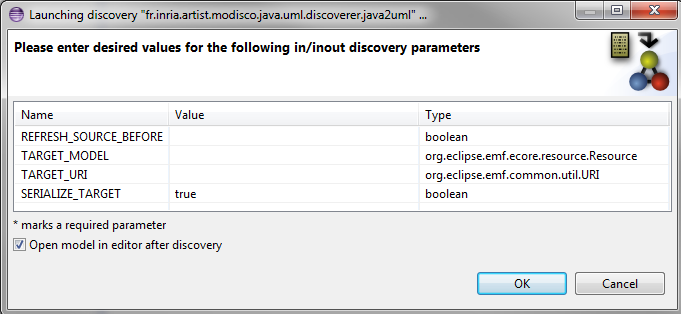
****

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.

#### Licensing information

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

### Component “Generic UML Activity Diagram Discoverer (behavioural part of the code) from Java source code – direct”

#### 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 [11]. 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).

#### 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 [8].*

#### 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 similar 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

#### 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).

#### 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.

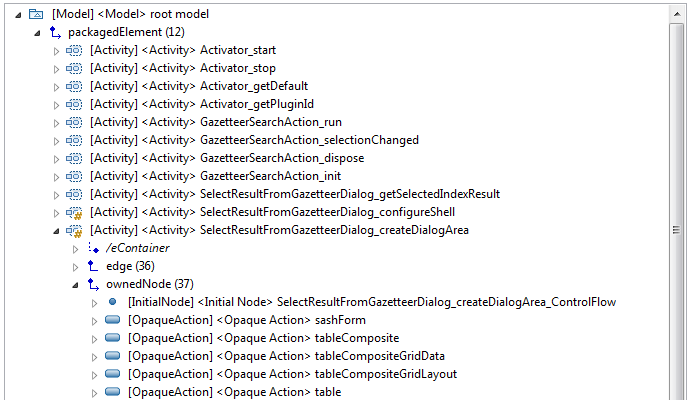


Figure 7 UML Activity Diagram model opened in the MoDisco Model Browser

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

#### Licensing information

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

### 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.

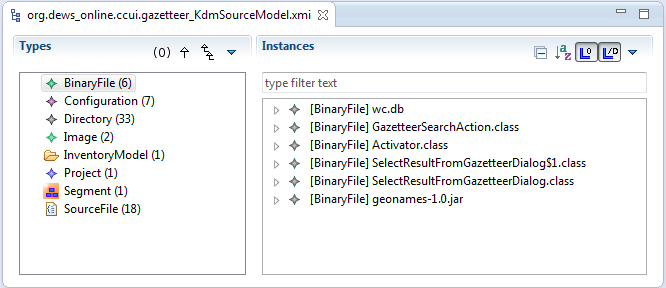


Figure 8 KDM Source model opened in the MoDisco Model Browser

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].

### Component “Generic UML Profile Discoverer from Java API/library”

#### Overview

Most modern programming languages nowadays provide a mechanism to express attributes over program elements, which leads 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 FindBugs 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 existing 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. Thereby, applied stereotypes capture platform-specific information, where the platform refers to the annotation-based Java library from the corresponding UML Profile was discovered. These stereotypes are exploited in the context of model understanding to produce dedicated model views or model slices. For instance, the UML Profile of the Java Persistence API (JPA) provides stereotypes to define entities (@Entity annotation) and embeddable entities (@Embeddable annotation) at the modelling level. Based on these stereotypes, a model slice can be computed that refers to the domain model of a reverse-engineered application. Furthermore, such stereotypes facilitate the improvement of reverse-engineered UML model from a qualitative perspective. Considering again the JPA-based UML Profile, associations annotated by the *Embedded* stereotype can be turned into composition relationships. In general, distinguishing between composition relationships and plain association relationships is not easily to achieve. From the perspective of the forward-engineering, UML stereotypes can be used to render UML models more specific. Stereotyped UML models leverage the development of code generators that produce platform-specific code. Such code generators are currently developed in the context of WP9. In this way, the UML Profile Discoverer acts as enabler for both reverse-engineering scenarios as well as forward-engineering scenarios by providing the required platform-specific UML Profile.

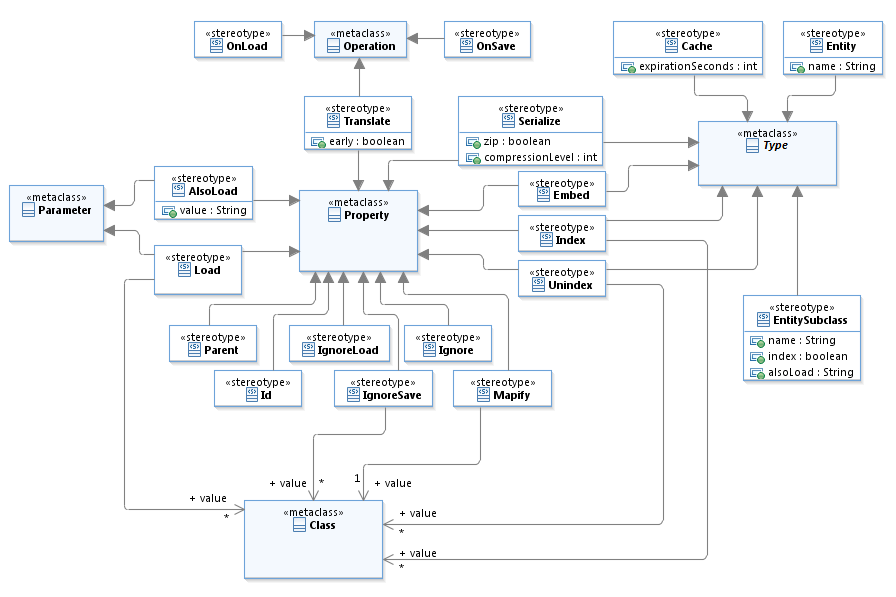


Figure 9 UML Profile model discovered from a Java library (*jar* file)

#### 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). These UML Profiles serve as foundation for annotating UML models with the same meta-information as Java code elements. Before the UML stereotypes (included in the profile) 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.

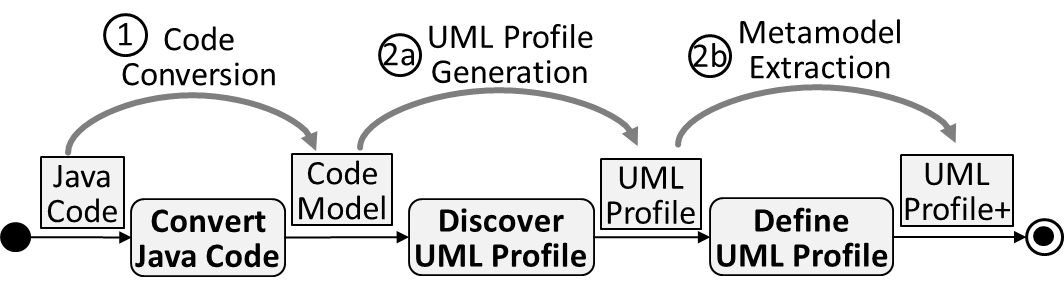


Figure 10 Conceptual Overview of the UML Profile Discoverer

#### Technical Specifications

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

#### 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/>.

#### 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.

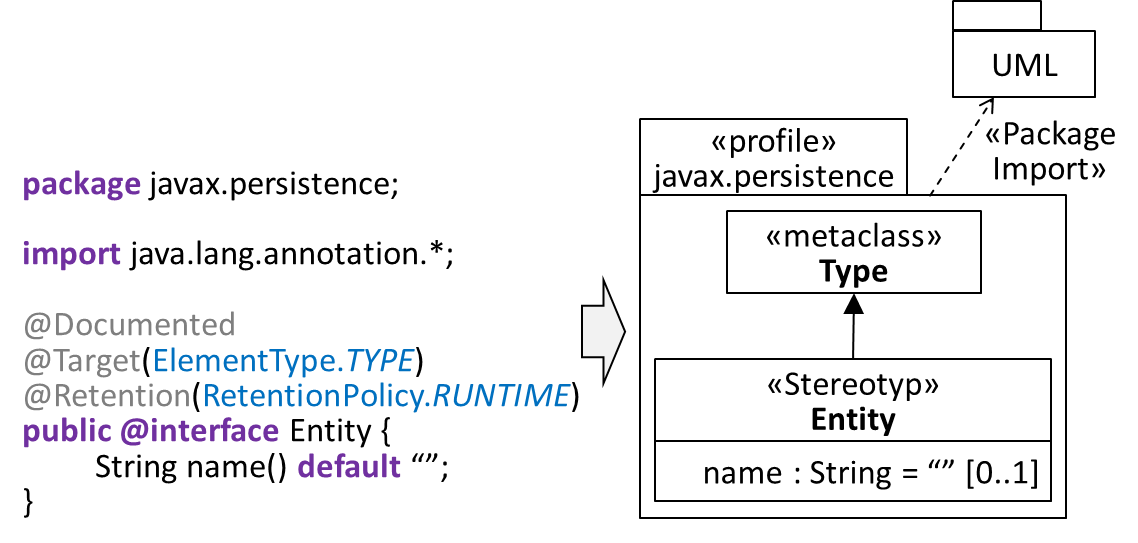


Figure 11 Example-based Java Petstore persistence layer

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. A demonstration video showcases the UML Profile Discoverer: <https://www.youtube.com/watch?v=c-PZA7sxdC4>. In this demonstration video, we show the necessary steps for automatically generating UML Profiles from annotation-based Java libraries. In fact, we demonstrate the UML Profile Discovery scenario according to the JPA. We explore the generated UML Profile by taking a closer look at the structure of the Entity stereotype, its extension to the UML metaclass Type and the required constraint to overcome the heterogeneity between Java's and UML's scope of Type. Finally, we briefly introduce the notion of the UML-Profile-Store as we exploit its capabilities the context of model understanding (see D8.3) as well as in the forward-engineering phase of the ARTIST methodology. Basically, the UML-Profile-Store is a collection of UML Profiles we have generated in the context of the ARTIST project.

From a technical perspective, they are available in terms of an Eclipse plugin, which exploits extensions points. In this way, developers can directly use the generated UML Profiles in the Papyrus UML model editor or access them in terms of a resource, which is helpful for the development of model transformations. For supporting frequently used annotation types at the modelling level without regenerating UML Profiles again and again, the UML-Profile-Store facilitates the reuse of already existing UML Profiles. The UML-Profile-Store covers currently 16 UML Profiles, comprising in total over 500 stereotypes that have been generated from annotation-based Java libraries, such as the Enterprise Java Beans (EJB), Hibernate, Java Persistence API (JPA) and Objectify.

#### Licensing information

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

## SQL Support

### Component “SQL Model Discoverer from SQL instructions”

#### Overview

This new component, packaged as a MoDisco discoverer, is a newly developed instructions-to-SQL Model injection implemented in Java, with the EMF [2] and the XText framework [7]. It also relies on an Ecore implementation of our SQL metamodel, generated via XText. It allows producing an SQL model from an initial set of SQL instructions. The obtained SQL model also only covers the ‘INSERT’ commands of a set of SQL instructions.

#### Functional description

This component currently considers:

* As input: An SQL instructions set (currently only the ‘INSERT’ commands are considered).
* As output: an SQL model, where all ‘INSERT’ elements contain the concerned table, columns and fields.

The actual conceptual mapping between SQL ‘INSERT’ commands and the SQL metamodel is specified by the instructions-to-SQL Model injection as previously introduced.

An overview of this mapping between elements from the SQL instructions and the SQL metamodel is presented in Table 24.

|  |  |
| --- | --- |
| **SQL instruction** | **SQL metamodel** |
| Set of INSERT commands | Model, containing the InsertSQL |
| INSERT | InsertSQL, containing Database, Tab, Column, and Field |
| [database] | Database, containing the name of the database |
| .[table] | Tab, containing the name of the table |
| ([column], ...) | Column, containing the name of the column |
| VALUES (field, ...) | Field, containing the value of the field |

Table 3 Mapping used in the SQL model injection

#### Technical specifications

The technical aspect of this component relies on a main EMF and XText injection and has been packaged as a MoDisco discoverer (named *eu.artist.migration.mdt.database.sql.discoverer*). The Eclispe EMF [2], Eclipse MoDisco [4] and Eclipse XText [1] technologies have also been used as a base for implementing this component.

This discoverer allows the selection of an SQL instructions set, and automatically launch the injection execution and serialization of the produced SQL model.

General information:

* Status: Alpha
* Release: 0.1

#### 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 XText SDK

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

#### User manual

To use the component, you can start from an SQL instructions set and then:

* Right-click on the package containing the SQL instructions set.
* Select the following menus *“Discovery / Discoverers / SQL Discoverer”.*
* Enter whether you want to serialize the output SQL model and manually set its location.
* Validate your choices in the window, and run the discovery process. The resulting SQL model will by default (i.e., if not specified differently) be saved at the same level than the SQL instructions set.

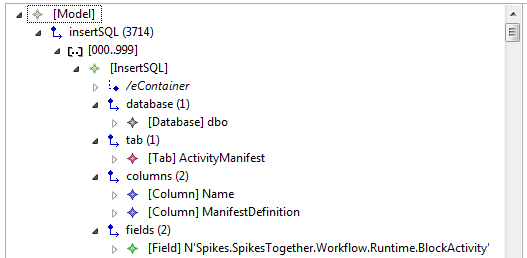


Figure 12 SQL model opened in the MoDisco Model Browser

The obtained SQL model can finally be navigated using the MoDisco Model Browser for instance (as shown on Figure 713).

#### Licensing information

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

### Component “Generic UML Object Diagram Discoverer from SQL model”

#### Overview

This new component, packaged as a MoDisco discoverer, is based on the chaining of the SQL instructions Model Discoverer with a newly developed SQL-to-UML (Object Diagram) model transformation implemented in ATL [1]. It also relies on an Ecore implementation of the OMG UML metamodel [10] and our SQL metamodel. It allows producing an UML Object Diagram model from an initial SQL instructions set (only passing by an intermediate SQL model). The obtained UML Object Diagram model covers the ‘INSERT’ commands by presenting the instantiation of the various classes (corresponding to tables in the database).

#### Functional description

This component currently considers:

* As input: An SQL instructions set, or eventually the SQL model representing the SQL instructions set (if the SQL model Discoverer has already been applied separately before)*.* An UML profile detailing (via stereotypes) which classes corresponds to the table of the database.
* As output: an UML Object Diagram, where each objects corresponds to each ‘INSERT’ commands.

The intermediate SQL instructions model being a direct representation of the content of the initial SQL instructions set, the actual conceptual mapping between SQL-UML profiled and UML (Object Diagram) is specified by the SQL\UML profiled-to-UML model transformation as previously introduced.

An overview of this mapping between elements from these two metamodels is presented in Table 23.

|  |  |  |
| --- | --- | --- |
| **UML metamodel profiled** | **SQL metamodel** | **UML metamodel (Object Diagram)** |
|  | Model | Model, containing the objects |
| Class | Tab | InstanceSpecification, without any link to another object, and containing the slot and the corresponding class |
| Property | Field | Slot, containing the Field value and the Property, for InstanceSpecification without any link to another object |
| LiteralString |  | Containing the Field value, for InstanceSpecification without any link to another object |
| Association | Tab | InstanceSpecification, with a link to another object, and containing the slot and the corresponding association |
| Property | Tab | Slot, containing the Tab reference and the Property, for InstanceSpecification with a link to another object |
| InstanceValue |  | Containing the Tab reference and the Property type, for InstanceSpecification with a link to another object |

Table 4 Mapping used in the SQL\UML profiled-to-UML Object diagram transformation

#### Technical specifications

The technical aspect of this component relies on a main ATL transformation (module) and has been packaged as a MoDisco discoverer (named *eu.artist.migration.mdt.database. sql.umlobject*). 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 SQL instructions 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

#### 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).

#### User manual

To use the component, you can start from an SQL instructions model, as previously obtained from the SQL Model Discoverer for instance (cf. section 3.1.12), and then:

* Right-click on the package containing the SQL instructions model and the UML profiled model.
* Select the following menus *“Discovery / Discoverers / SQL to UML Object Discoverer”.*
* 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 SQL instructions model.

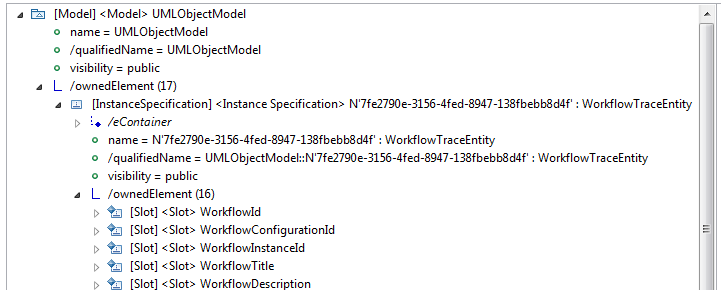


Figure 133 UML Object Diagram model opened in the MoDisco Model Browser

The obtained UML model can finally be navigated using the MoDisco Model Browser for instance (as shown on Figure 713).

#### Licensing information

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

## .NET Support

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

#### Overview

With this component built on top of Sparx 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 discover 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.

#### 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.

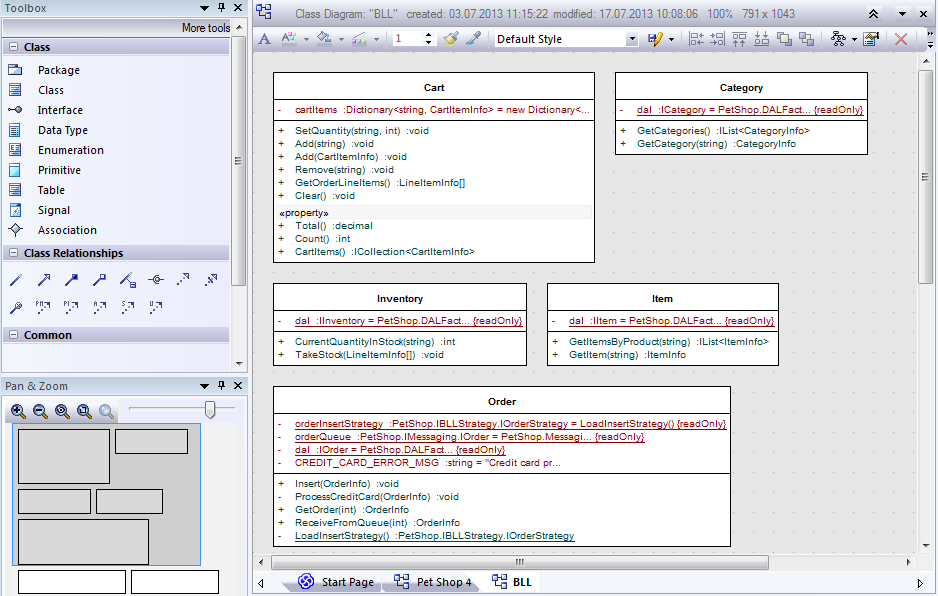


Figure 144 UML Class Diagram model opened in Sparx Enterprise Architect

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).

#### Technical specifications

Sparx 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, Team Foundation Server (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).

TO BE IMPROVED BY DESCRIBING MORE SPECIFICALLY THE COMPONENT ITSELF?

#### Download and installation instructions

This component is contained within the standard installation of Sparx Enterprise Architect. Sparx 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 Sparx 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 Sparx Enterprise Architect versions.

The download can be obtained directly from [14]. 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).

#### User Manual

**Import of Source Code from a Source Code Folder**

Within a Sparx 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 Sparx 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).

#### Licensing information

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

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

#### 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 Sparx 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.

#### Functional description

Sparx 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.

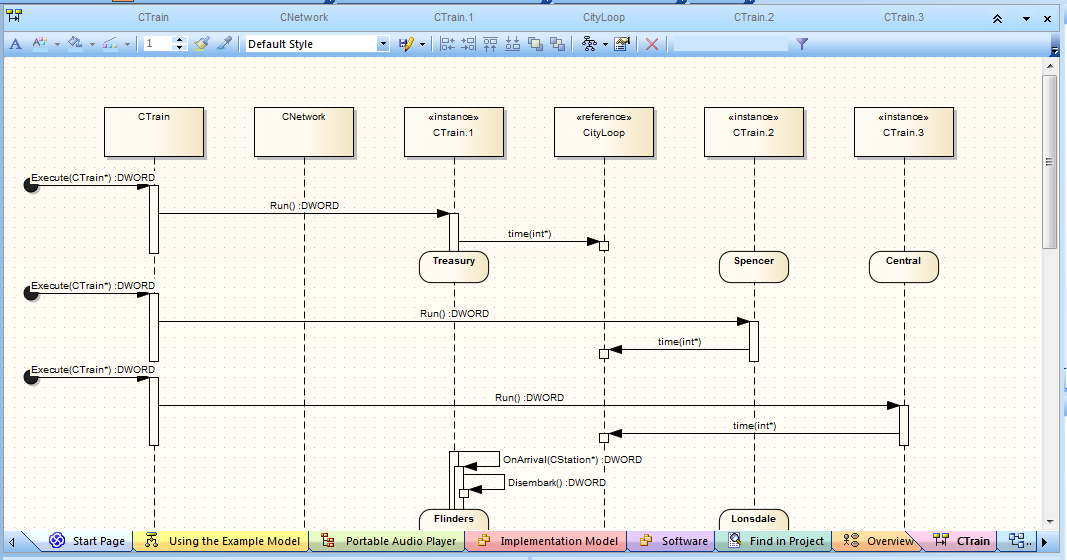


Figure 155 UML Sequence Diagram model opened in Sparx Enterprise Architect

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

#### 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.

#### 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 3.3.1.4).

#### User Manual

Before using the MDDE, check if you are using the correct edition that includes it: Sparx 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 3.3.1).

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

#### Licensing information

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

### Component “Generic UML Class Diagram Discoverer from Microsoft library”

#### Overview

With this component built on top of Sparx 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 3.3.1), 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 3.3.1.1).

#### 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 16).

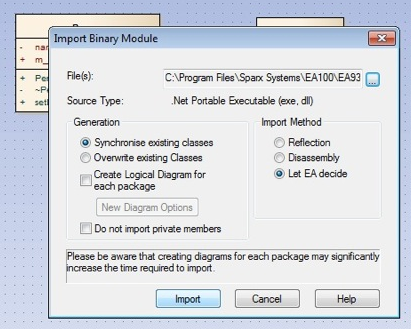


Figure 166 Options for discovering a UML Class Diagram model from a Microsoft library

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

#### 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 3.3.1.3).

#### 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 3.3.1.4).

#### User Manual

Within Sparx Enterprise Architect 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.

#### Licensing information

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

### Component “SharePoint Data Model Discoverer from SharePoint database”

#### Overview

Customizations or extensions in Microsoft SharePoint [12] 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).

#### 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).

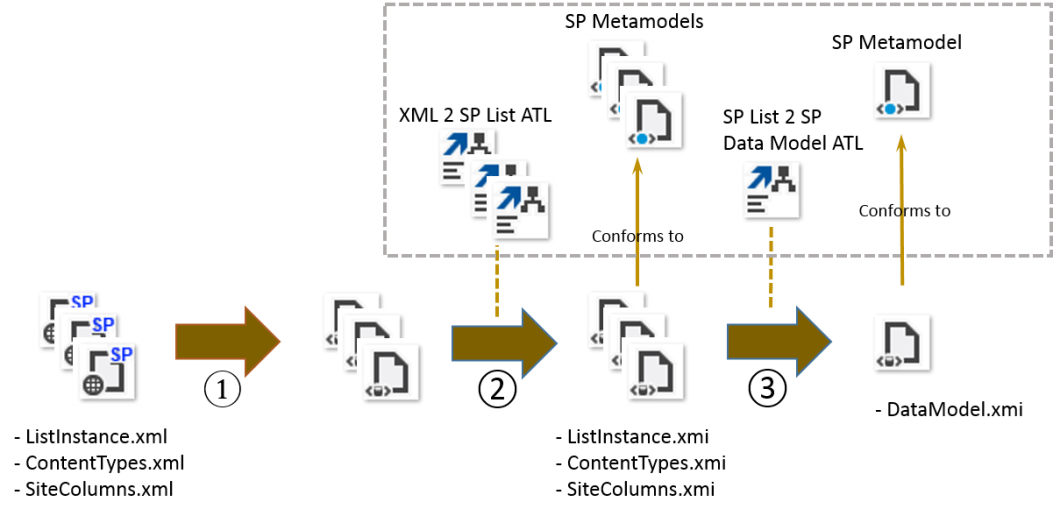


Figure 17 SharePoint Data Model Discoverer: Architecture Overview

In step (1), XML models are discovered from XML files conforming to the SharePoint schema definitions [12]. This is done using the Generic XML Model Discoverer as introduced in this document. 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.

#### 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

#### 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).

#### 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.

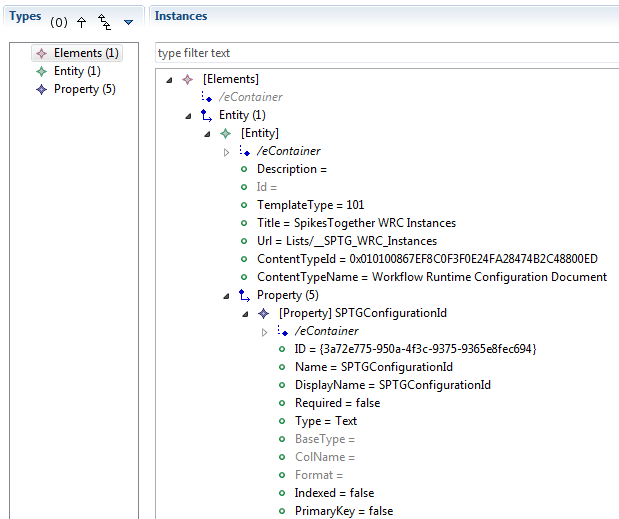


Figure 18 SharePoint Data model opened in the MoDisco Model Browser

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 8).

#### Licensing information

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

## Generic Support

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

#### 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.

#### 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 ).

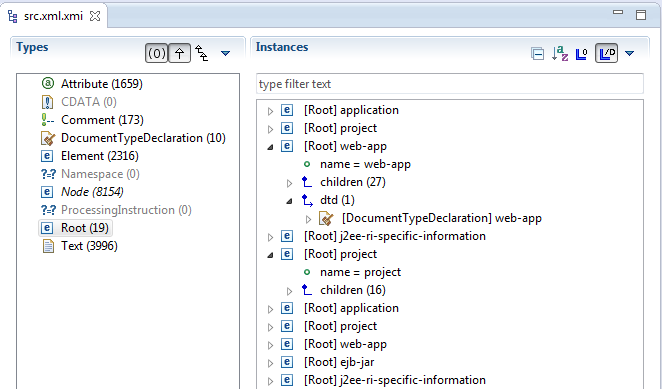


Figure 19 XML (Aggregated) model opened in the MoDisco Model Browser

#### 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.

#### 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).

#### 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 9 for seeing an example).

#### Licensing information

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

## Component “Reusability Trace Tool”

### Overview

The currently available tools and processes for Reverse and Forward Engineering do not provide dedicated mechanisms for selecting certain parts of the source code and then transfer them directly to the target system. The migration or modernization process is done through 1) reverse engineering techniques generating models and 2) sets of M2M and then M2T transformations to generate the target code from these models. However, in cases there is no change of programming language, sometimes it is necessary (and more effective) to transfer some fragments of the source software source code to the target system without executing the full transformation process.

The objective of the “Reusability Trace Tool” (or RTT) is 1) to provide a first automatic selection of pieces of code that can be transferred directly to the target system and 2) to permit the user to participate in the process and take actual decisions.

### Functional description

An Eclipse-based tool has been developed allowing reusing the input system’s code in a non-intrusive way, i.e., without modifying the models generated all along the process.

The “Reusability tool” will allow the user to set the criteria for selecting the code he/she thinks appropriate to reuse. After setting the criteria to be applied, an algorithm will be executed. This algorithm will use metrics established in the scope of WP5 and, when finished, the user will see on a window the set of code snippets automatically proposed to be reused (class and method names). Through this result window, the user will be able to decide whether to accept the automatic proposal or to make some changes by selecting/deselecting the correspondent methods names (via check boxes). The finally selected method names will be registered in a configuration file.

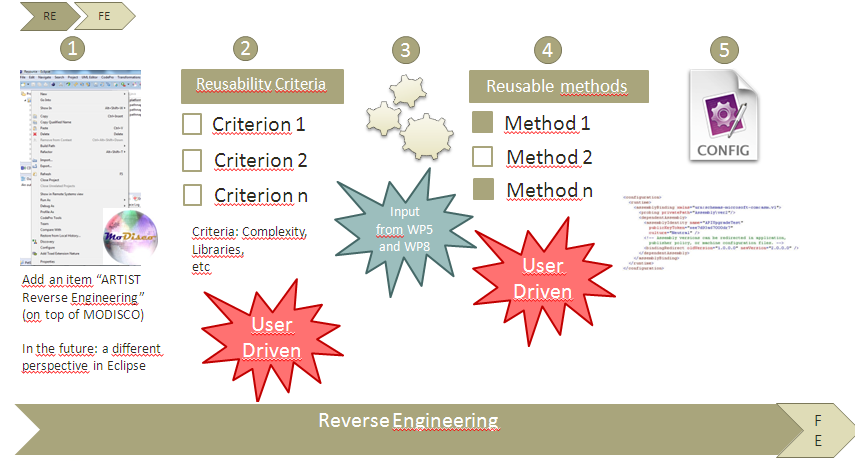


Figure 20 Traceability in Reverse Engineering with RTT (TODO – CHECK TYPOS ON FIGURE)

Inside forward engineering tools (WP9) integrated in Eclipse, it will be implemented a mechanism that will allow the user to rewrite the code generated with M2M and M2T transformations by accessing directly the source software code and the list of reusable methods in the previously produced configuration file.

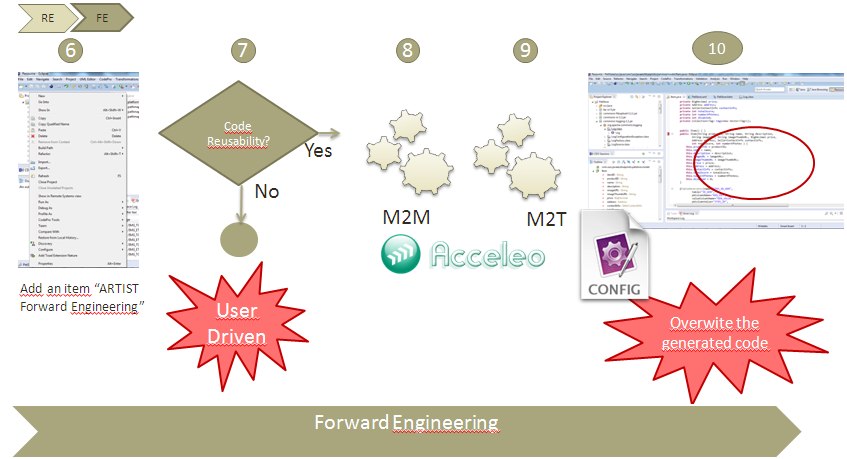


Figure 21 Traceability in Forward Engineering with RTT

### Technical specifications

The tool has been created as an Eclipse plugin contributing to the workbench, by using the SWT library and WindowBuilder as the Java GUI designer.

Next are detailed the main products and versions used in the development:

* Eclipse Kepler SR1 on Windows
* Java SE-1.7
* Window Builder 1.6.1

SWT is an open source widget toolkit for Java designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented. WindowBuilder is a powerful and easy-to-use bi-directional Java GUI designer composed of SWT Designer and Swing Designer that makes easy creating Java GUI applications without spending a lot of time writing code. Using the WYSIWYG visual designer and layout tools allows building simple forms to complex windows; the Java code will be generated automatically. It permits also easily adding controls using drag-and-drop, adding event handlers to those controls, changing properties of them by using a property editor, internationalising an application and much more.

### Download and installation instructions

This component requires the following software to be installed:

* Eclipse Kepler SR1
* RTT plugin (reusability.jar) can be downloaded from the public repository (cf. section 2.5).

It is also needed to copy a configuration file called *config.xml* to the file system where the plugin is executed.

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

### User manual

This section describes how can be used the “Reusability Tool” from a practical point of view. First, the user has to execute the plugin by clicking on the new menu option.

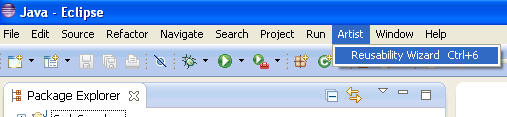


Figure 172 RTT activation in Eclipse

After doing this, a modal window is opened.

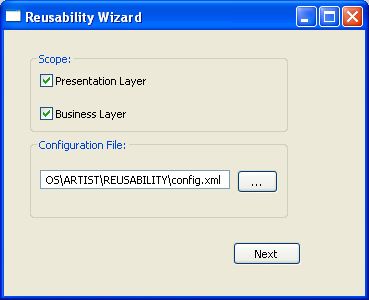


Figure 183 RTT wizard

Here, the user has to set the reusability scope. In the first approach we have decided that it could be “Presentation level“ or  “Business Level”. In case “Presentation Level” is selected, next window is opened

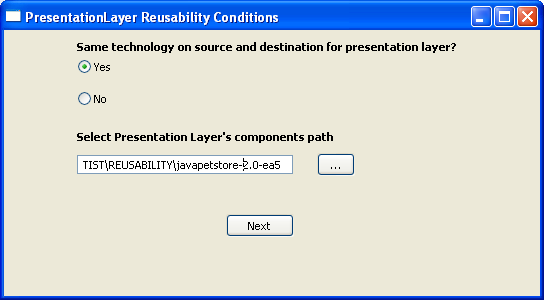


Figure 194 RTT wizard for Presentation layer

Here it would be possible to select the path where presentation components are located in case same technology is used both in source and target systems. In case of using different technologies at presentation layer (for example *jsp* in one site and *php* in the other), it would be impossible to reuse directly  any part of code at this level. In case “Business Level” is selected, next window is opened.

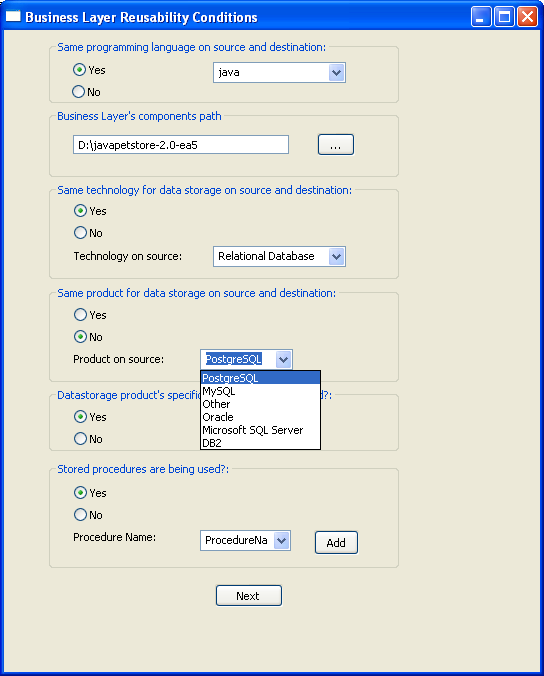


Figure 205 RTT wizard for Business layer

Through this window, the user can establish the conditions to automatically select the pieces of code (methods) that could be reused directly. After the conditions are established, next window is opened.

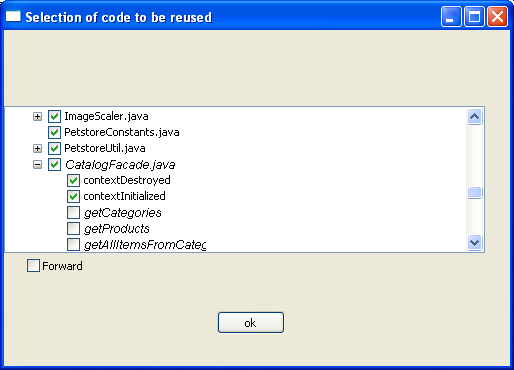


Figure 216 RTT wizard for reusable code selection

This screen contains a tree with the list of all the components and methods that could be reused directly according to the conditions established previously. Despite of this automatic selection, the user will still have the choice to select and unselect any component or method he/she thinks appropriate.

After this, when pushing the “Ok” button, a configuration file (xml or json) is generated with the relation of all the components and methods to be reused directly (this way it will be possible to access the code directly as it is on the source during the forward engineering process).

### Licensing information

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

# Conclusion and Future Work

TODO – Update/extend to reflect the updated content (as well as new sections) and current situation at M24!

The present deliverable is about introducing the second 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 software artefacts composing the existing systems to be migrated (potentially coming either from the Java EE or .NET technologies), 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 the first year of the ARTIST project, the foundations have been set in order to build the initial version of the ARTIST Model Discovery Toolbox. As a summary, the work performed during this first period had 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.

During the second year of the ARTIST project, further developments have been realized in order to consolidate and extend the initial version of the ARTIST Model Discovery Toolbox and to come up with this second version. As a summary, the work performed during this second period has mainly consisted in:

1. Improving several of the components according to the feedback collected during their concrete applications in the context of ARTIST use cases. This has notably been made in collaboration with T8.3 (Model Understanding) and WP9 as main “consumers” of the discovered models.
2. Developing new discovery components for the identified missing support or requirements expressed by the consortium. Thus, the support for model discovery in C# is being notably improved. On the other hand, the model discovery in Java has been enriched with new support. Finally new generic UML discoverers (e.g. from different databases) have been/are being developed.
3. TO BE UPGRADED FOR M24!

The current version of the toolbox is mainly resulting from 1) the various experiments conducted during the first year of the project on the commonly agreed PetStore example and 2) the application of the different components in particular project’s use cases. It is important to mention that some of the newly developed components, relying on previously existing components or built from scratch, are still work in progress. Thus, the objective for the last year of the project (and for the delivery of the third version of the ARTIST Model Discovery Toolbox) is twofold:

* Still continue to consolidate the components prototyped/developed during the two first years and already provided as part of the second version of the toolbox. The application of the developed discoverers in the use cases defined within the project is providing valuable feedback to improve them and make them more efficient.
* 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:
  + An improved C# model discoverer (.NET-specific support).
  + A more advanced support for the discovery of the behavioural aspects of the source code (both for Java and C#).
  + A more advanced support for traces in both Java and C# technologies.
  + Potentially other components to be determined during the running of the project.
  + TO BE UPGRADED FOR M24!

# 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. Eclipse XText tool. Available at: <http://www.eclipse.org/Xtext/>
8. OMG ADM Task Force. Homepage: <http://adm.omg.org/>
9. OMG fUML specification. Available at: <http://www.omg.org/spec/FUML/>
10. OMG KDM specification. Available at: <http://www.omg.org/technology/kdm/>
11. OMG UML specification. Available at: <http://www.uml.org/>
12. SharePoint schema definition. Available at: <http://msdn.microsoft.com/en-us/library/ms476062.aspx>
13. Sparx Enterprise Architect toolset. Available at: <http://www.sparxsystems.com/products/ea/>
14. Sparx Enterprise Architect (trial version) download. Available at: <http://www.sparxsystems.com/products/ea/trial.html>