(ITEA 2 – 13017)

Enabling of Results from AMALTHEA and others
for Transfer into Application and
building Community around

Deliverable: D 1.2

Specification of Traceability concepts

Work Package: 1

Continuous Design Flow and Methodology

Task: 1.4 - 1.6

Specification of the traceability concept for requirements
Development of reference implementation

Document Type:	Deliverable	**Classification:**	Public
Document Version:	1.0	**Contract Start Date:**	01.09.2014
Document Preparation Date:	01.08.2016	**Duration:**	31.08.2017

ITEA 2
INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

EUREKA

History

Rev.	Content	Resp. Partner	Date
0.1	Set up Document	Salome Maro	August 1st, 2016
0.2	Initial Content	Salome Maro	August 5th, 2016
0.2	Document Initial Review	Jan-Philipp Steghöfer	August 15th, 2016
0.3	Document Review by WP1 Members	Jan-Philipp Steghöfer	August 20th, 2016
0.4	Preparation of Final Document	Salome Maro	Sept 19th, 2016
0.5	Final Document Review	Jan-Philipp Steghöfer	Sept 23rd, 2016
1.0	Final Document Submission	Jan-Philipp Steghöfer	Sept 23rd, 2016

Capra Help Contents

User Guide

Introduction — What is Capra?

Capra is a dedicated traceability management tool that allows the creation, management, visualization, and analysis of traceability links within Eclipse. Traceability links can be created between arbitrary artifacts, including all EMF model elements, all types of source code files supported by the Eclipse Platform through specialized development tools, tickets and bugs managed by Eclipse Mylyn, and all other artifacts for which an appropriate wrapper is provided. Capra is highly configurable and allows users (in a company or project) to define link types that are useful to them.

Compared to other similar projects which may have similar features, Capra is not a modeling tool or a tool for requirements management. All functionality is focused on providing traceability capabilities, i.e., the ability to create and visualize links between artifacts modeled in different domain-specific languages. This allows the architecture to be highly modular and the tool to be extremely customizable.

Getting Started

This section describes the prerequisites to run Capra and how to install the tool.

Prerequisites

Before downloading and using Capra, first download a distribution of the Eclipse Modeling Environment (Eclipse V 4.5 (Mars)) and make sure you have the following installed:

- PlantUML: Use the nightlies update site in Eclipse's "Install new software..." feature. Version 1.1.11 or higher should be installed through this link. Older versions advertised on the website will not work! It might also be necessary to install Graphviz binaries on your system to view the visualisation of traceability links.
- Xcore: Install through Eclipse's "Install new software..." feature. Select the Mars distribution "http://download.eclipse.org/releases/mars/" in the work with field.
- Mylyn: Install the "Mylyn Builds Connector: Hudson/Jenkins" through Eclipse's "Install new software..." feature
- C/C++ Development Tools: : Use Eclipse's "Install new software..." feature
- Java Development Tools: Use Eclipse's "Install new software..." feature
- Xtend: Install through the Eclipse Market Place
- Papyrus: Install through the Eclipse Market Place

Installing Capra

Capra can be installed either through an update site or manually.

Installing Capra through an update site

- Go to the Capra Github repository.
- Click on "Releases" (Please note that the current release is a Pre-release and the official release will be available once the tool is an Eclipse project)
- Download the file `Capra_v.0.1.zip` (Note the location of your download)
- Open your Eclipse environment
- Make sure that you have all the dependencies listed above installed
- Click on Help >> Install New Software
- Click on Add and select Archive
- Select your zip file and click Add
- You should see the features that Capra has grouped into several categories as shown in Figure 1.
- To get a minimum version of Capra working, install all the features in Core and at least one artifact wrapper. This means that if you for instance install only the EMFHandler, then you will only be able to create traceability links between model elements contained in EMF models. It is recommended to install all available artifact handlers.
- Install the features in the visualization and notification categories in order to get their functionality working.
- Restart Eclipse
- Go to perspectives and switch to the Capra perspective
- Now you can create traceability links as described in Creating Traceability Links.

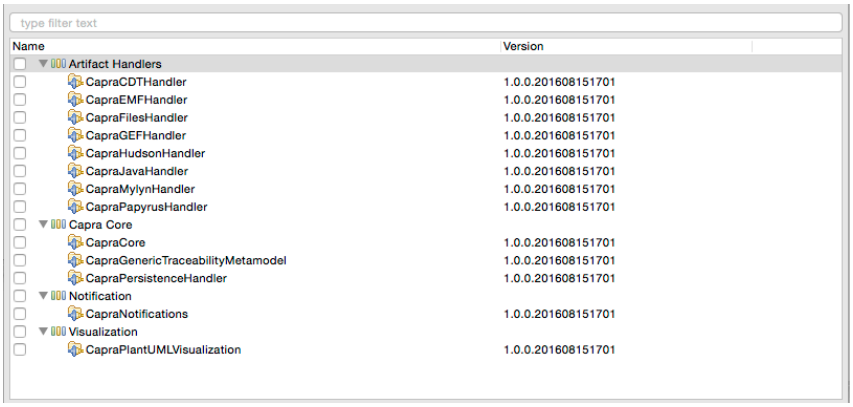


Figure 1: Installing Capra from the Updatesite

Installing Capra manually with Git

- Open your Eclipse Environment
- Go to File >> Import and select Git >> Projects from Git
- Use the Git repository and import all available projects to your workspace
- Build your workspace
- Make sure that all the projects have no errors.
- Click on Run >> Run Configurations and create a new Eclipse Application Configuration
- Select your running workspace
- Click Finish
- Once the new workspace opens, create or import projects that you want to use to create traceability links
- Go to perspectives and switch to the Capra perspective
- Now you can create traceability links as described in Creating Traceability Links.

Traceability Concepts

In this section, important concepts for traceability are described.

Definition of traceability

Traceability can be defined as the ability to relate different artifacts created during the development of a software system. Traceability allows creating and using links between system and software development artifacts. For instance, this allows connecting the origin of a requirement with its specification, the design elements that address its specification, the code that implements these design elements, and the acceptance tests that check if the requirement has been achieved.

What is a traceability link?

The connections between different artifacts in a software development environment are called traceability links. A traceability link can connect two or more elements to imply that there is a relationship between these elements.

Types of traceability links

Depending on the requirements of a domain, traceability links can have different types. There are three different categories that can lead to different types of traceability links: the shape of a link, the semantics of a link, and the direction of a link.

- Link Shape: A link can have a source and target which means that you can only use it to connect two artifacts, or it can be of N-nary type, meaning that you can connect more than two artifacts using the same link.
- Link Semantics: Traceability links can carry different semantics. The semantics of a link can be implied through the link name, e.g., “satisfies”, “implements”, “tests”. It is also possible to enforce “technical” restrictions on which kind of artifacts a link can connect. For example, a “tests” link type can only allow connecting a test to a requirement. Link semantics are defined through a traceability metamodel and are configurable in Capra.
- Direction of a link: A traceability link can be either a directed link or a bi-directional link. A directed link has a “source” and “target(s)” relationship where the link is from the source and points to the target(s). A directed link also means that one can navigate from “source” to target(s) but not vice versa. A bi-directional link means that the link has no semantics in the direction and one can navigate from any connected artifact to the other connected artifact(s).

What is an artifact?

Artifacts, and in particular software development artifacts refer to the resources that are either created or used as an input by a software development activity. For instance, the requirements elicitation activity produces artifacts known as requirements. Artifacts can be of different types, such as a requirement, a model element, a line of code, or a test case.

What is an artifact handler?

As previously mentioned, there are different types of artifacts that can exist in a software development environment. However, Capra stores the traceability links in form of an EMF model. This means that, in principle, only EMF artifact types can be supported. To support other artifact types, there is a need to create EMF representations of the artifacts. This is what an artifact handler does. It creates an EMF representation of non-EMF artifacts. The representations are known as “artifact wrappers”. For example to be able to link to Java Code, an artifact handler for Java needs to be created. For details on how to add new artifact handlers to the tool, refer to Adding a new artifact handler.

Practical Examples

To demonstrate features of Capra, we take an example of the development of a Heating, Ventilation and Air Conditioning (HVAC) System. The resources can be dowloaded here. The project contains the following artifacts:

- A set of requirements which are written in ReqIF format.
- A feature model describing the different features that can form several products of the HVAC system.
- State machines used to describe the behavior of the system
- Generated code from the state machines
- Test cases defined as Java Unit tests and
- Issues/Tasks/Tickets reported as development progresses which are stored on a Trac server

These artifacts are shown in the figure below in the context of the development environment of the HVAC system.

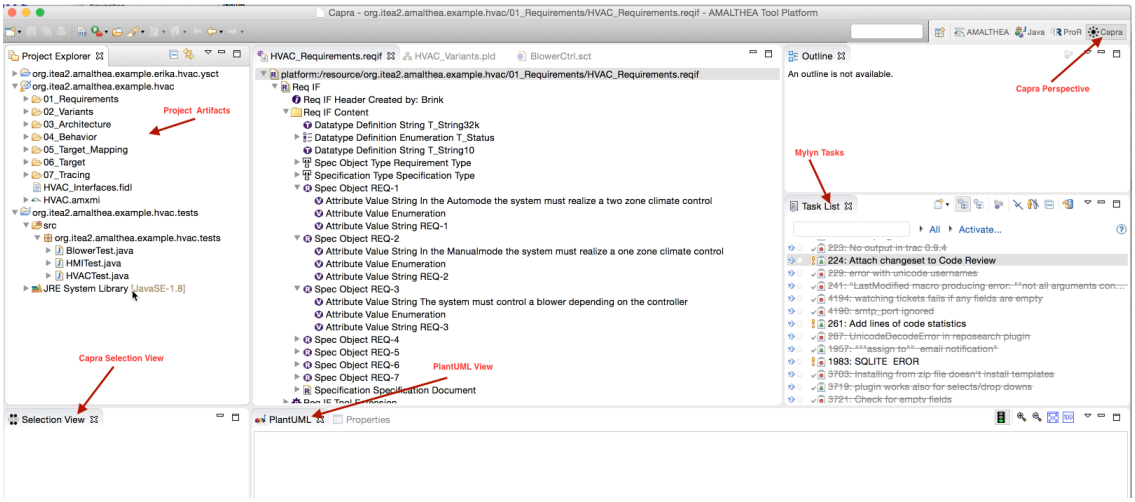


Figure 2: Resources for the HVAC system

Creating Traceability Links

Capra provides the functionality to create traceability links between different artifacts as long as artifact handlers for those artifact types are available. The current version supports tracing to EMF models, Java code (up to method level), C/C++ code (up to function level), Task tickets from ticketing systems supported by Mylyn, arbitrary files (such as PDF or word), Test executions (Hudson and Jenkins), Papyrus models, and Capella models.

To show how traceability links can be created, we continue with the HVAC example and its artifacts as described above. Our aim is to establish the following links:

1. A link from a requirement to a feature representing the requirement in the feature model
2. A link from a feature to the state machine that describes the behavior of the feature
3. A link from a state machine to test case that tests the behavior described by the state machine
4. A link from a requirement to a PDF document that describes safety issues that need to be considered
5. A link from a task ticket to a requirement that is associated with the ticket

The procedure to create the above links is described below.

A link from a requirement to a feature representing the requirement in the feature model

1. To start, open the requirements (`HVAC_Requirements.reqif` file), with the “Sample Reflective Ecore Editor” view.
2. Expand the model to see the requirements. Drag Req 3 and drop it in the Selection view.
3. Next, open the feature model (`HVAC_Variants.pld`), drag and drop the feature named “Blower” into the selection view as well as shown in the figure below.

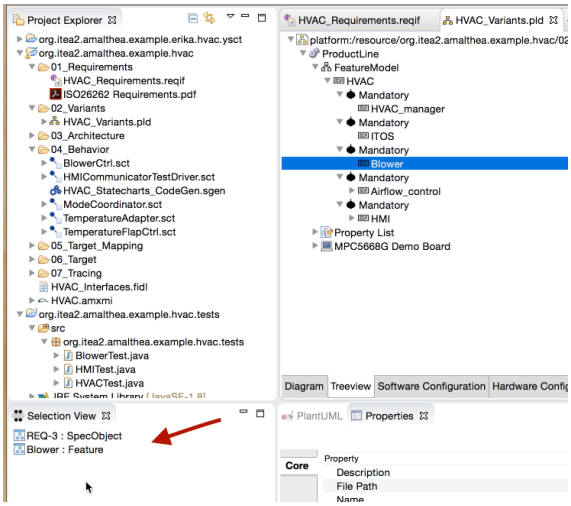


Figure 3: Capra Selection View

4. Right click on the selection view and click on “Create Trace”.

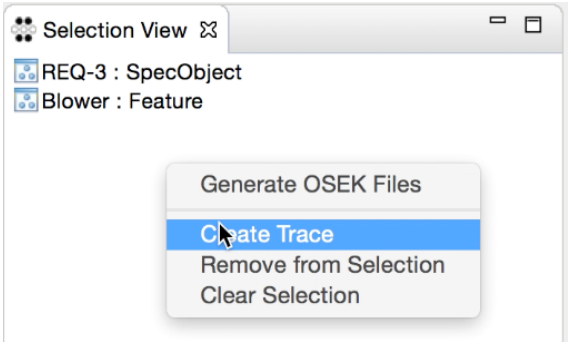


Figure 4: Creating a traceability link

5. A pop up window will appear, showing the types of links that can be created based on the selected element.
6. Choose a traceability link type to create (in our case “RelatedTo”) and click OK.

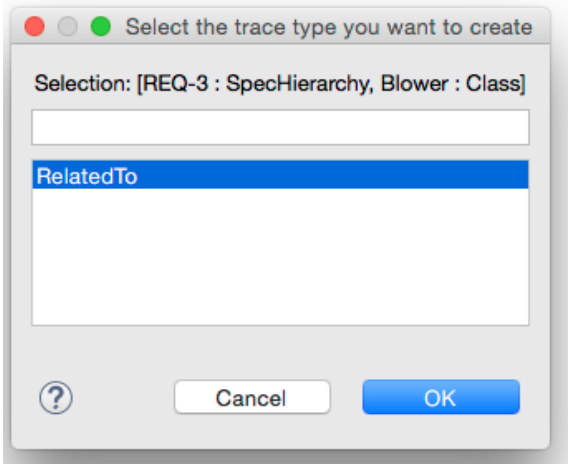


Figure 5: Creating a traceability link of type “RelatedTo”

Since this is our first traceability link, a new folder will appear in the workspace with the name `__WorkspaceTraceModels`. This folder contains your trace model which contains the traceability link we just created and an artifact wrapper model which contains EMF representations of artifacts that are not in EMF format. In our case the artifact model should be empty since the artifacts we used to create the traceability link are all EMF elements. The trace model (`traceModel.xmi`) should contain only one traceability link.

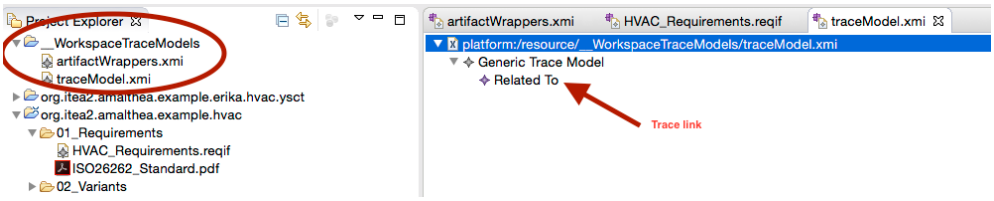


Figure 6: A traceability link of type “RelatedTo”

7. To remove the elements from the selection view, there are two options:
    1. Right click on an element and select “Remove from Selection”. This will only remove the selected element.
    2. Right click anywhere on the selection view and click on “Clear selection”. This will remove all elements in the selection view.

A link from a feature to the state machine that describes the behavior of the feature

8. From the feature model, drag and drop the feature named “Blower” into the selection view.
9. Open the `BlowerCtlr` state machine and drag and drop the parent element to the selection view.
10. Right click on the selection view and click on “Create Trace”.
11. A pop up window will appear, showing the types of links that can be created based on the selected elements.
12. Choose a traceability link type to create (in our case “RelatedTo”) and click OK. Expand the `__WorkspaceTraceModels` project and open the trace model (`traceModel.xmi`). You will notice that a second traceability link has been created.

A link from a state machine to test case that tests the behavior described by the state machine

13. Clear the selection view.
14. Again, Open the `BlowerCtlr` state machine and drag and drop the parent element to the selection view.
15. Expand the project containing the tests, expand the `BlowerTest.java` file to reveal the BlowerTest class. Drag and drop this class to the selection view.
16. Right click on the selection view and click on “Create Trace”.
17. A pop up window will appear, showing the types of links that can be created based on the selected elements.
18. Choose a traceability link type to create (in our case “RelatedTo”) and click OK.

A link from a requirement to a PDF document that describes safety issues that need to be considered

19. Clear the selection view.
20. Expand the folder containing the requirements to reveal its contents.
21. Open the requirements (`HVAC_Requirements.reqif` file), with the “Sample Reflective Ecore Editor” view.
22. Expand the model to see the requirements. Drag Req 3 and drop it in the Selection view.
23. Select the `ISO26262 Requirements.png` file from the Project Explorer and drag and drop it to the selection view
24. Right click on the selection view and click on “Create Trace”.
25. A pop up window will appear, showing the types of links that can be created based on the selected elements and the definition of the traceability metamodel. In this example the traceability metamodel has only one traceability link type which is called “RelatedTo”.
26. Choose a traceability link type to create (in our case “RelatedTo”) and click OK.

A link from a task ticket to a requirement that is associated with the ticket

27. Clear the selection view
28. Drag Req 3 and drop it in the Selection view.
29. From the `Task List` view, where the tasks from the Trac server are listed, select one task and drag and drop it to the selection view
30. Right click on the selection view and click on “Create Trace”.
31. A pop up window will appear, showing the types of links that can be created based on the selected elements.
32. Choose a traceability link type to create (in our case “RelatedTo”) and click OK.

Visualizing Traceability Links

Capra offers two ways in which you can visualize the traceability links that you have created. These are the Graphical view where the artifacts are shown as nodes and the links as edges in a graph and the Matrix view where artifacts are arranged in rows and column with an “x” mark in the cells to indicate a traceability link between the artifact in the column and that in the specific row.

Graphical view

To view the traceability links related to an artifact and the connected artifacts, simply select the artifact while in the “Sample Reflective Editor” View. The “Plant UML View” needs to be open as well.

The graphical view allows you to explore directly connected elements or transitively connected elements. To use the latter functionality, click on the downward arrow on right hand corner of the Plant UML View and click on “Toggle Transitivity”. This enables you to move from viewing only directly connected elements to the selected element, to viewing all the transitively connected elements. Use the same button to return to the previously active view.

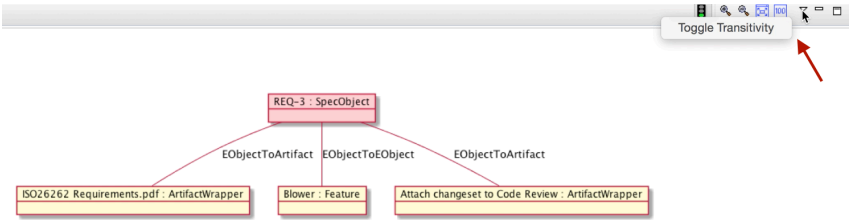


Figure 7: Toggle Transitivity

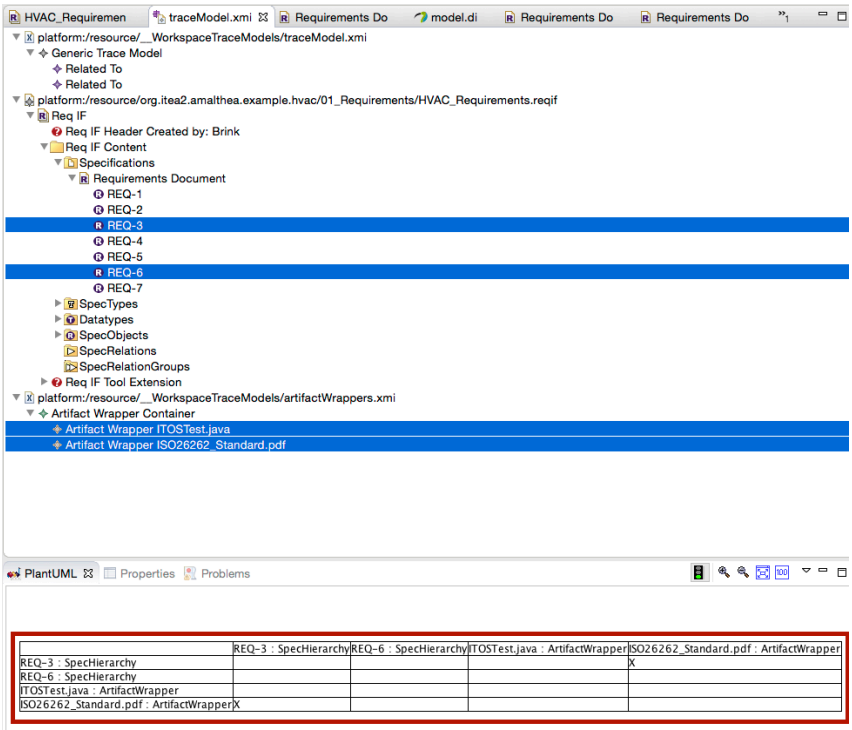
Traceability matrix

The traceability matrix can be created by selecting at least two model elements when the “Plant UML View” is open. This will list all the model elements as rows and columns and an “x” mark will appear to show that there is a traceability link between two elements. For instance, the picture below shows the resulting matrix when selecting Req3 and the artifact wrapper representing the PDF document.

	ISO26262_Standard.pdf : ArtifactWrapper
REQ-3 : SpecHierarchy	X

Figure 8: Matrix View as a result of selecting two elements

Selecting more than two model elements expands the matrix into a square matrix with same elements listed vertically and horizontally.



	REQ-3 : SpecHierarchy	REQ-6 : SpecHierarchy	ITOSTest.java : ArtifactWrapper	ISO26262_Standard.pdf : ArtifactWrapper
REQ-3 : SpecHierarchy				X
REQ-6 : SpecHierarchy				
ITOSTest.java : ArtifactWrapper				
ISO26262_Standard.pdf : ArtifactWrapper	X			

Figure 9: Matrix view as a result of selecting more than two elements

Detecting and Fixing Inconsistencies

Traceability links need to be updated as the artifacts they connect evolve. Capra provides a feature to notify users when these artifacts change and to give suggestions on how the traceability links can be changed accordingly. The suggestions are offered as quick fixes to the user and if the user wants to make the changes suggested by the quick fix, the fix can be applied automatically by clicking on it. Currently Capra uses the Eclipse Notification Framework to detect changes and can capture rename, move, change and delete actions made on artifacts that have traceability links.

The problems detected by Capra are shown in the `Problems View` with a type “Capra problem”. We demonstrate the use of the `Problems View` and quick fixes using our practical example of the HVAC project.

1. Go to the “Project Explorer” and expand the project containing test cases.
2. Delete the file `BlowerTest.java`.
3. Look at the `Problems view` and you will see a warning with a type “Capra Problem”. The issue tells you that there is a traceability link

that points to a file named `BlowerTest.java`, but that file has been deleted.

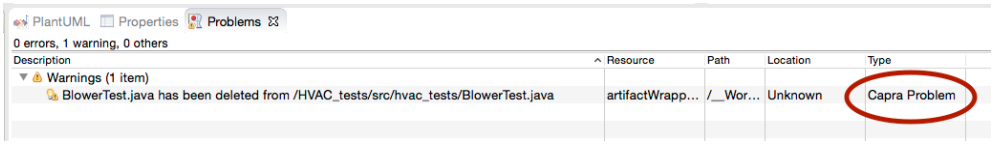


Figure 10: Problem view showing Capra errors

4. Right click on the warning and select Quick fix.
5. A window will appear showing the quick fixes options available. In this case there is one option, which is to delete the traceability link related to `BlowerTest.java` file.

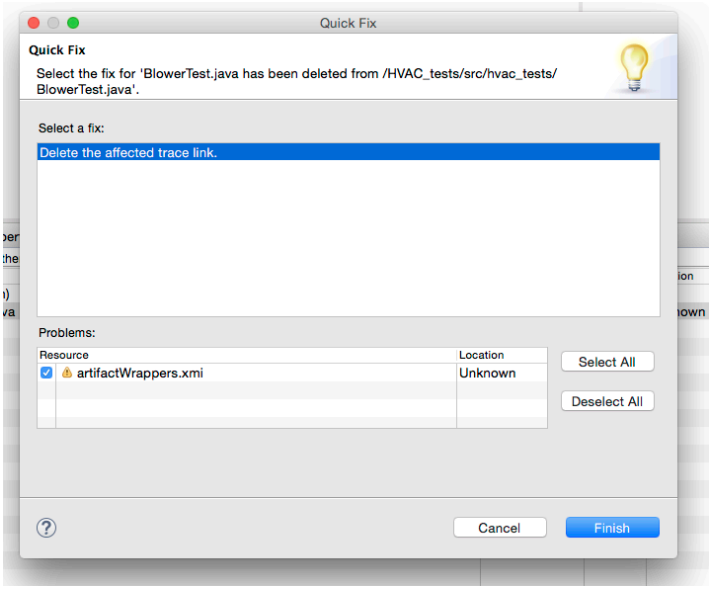


Figure 11: Quick Fixes

6. Click Finish.
7. Notice that the traceability link has been deleted and the warning disappears.

Analyzing change impact

In this section, we describe scenarios in which Capra can be used to facilitate change impact analysis. Change impact analysis allows to evaluate the effect a change to an artifact will have on other artifacts. Using the HVAC example, assume that the customer requests a change on the requirement `REQ-3`. Before such a change is made, it is important for the company to know which other artefacts will be affected. With Capra, this can be achieved by selecting `REQ-3` and, using the visualization, reviewing all other artefacts that are related to `REQ-3` too.

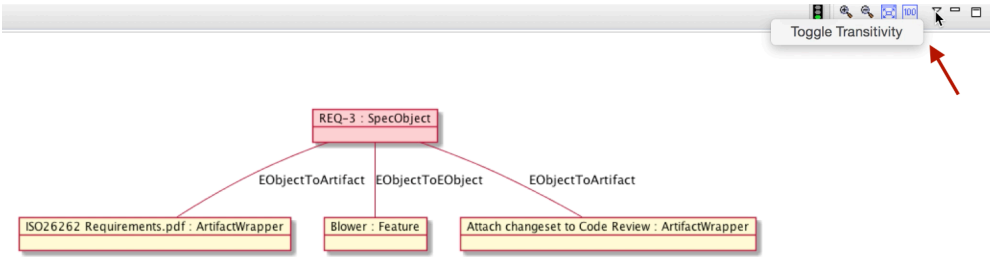


Figure 12: Capra graphical view showing directly connected elements

For further analysis, clicking on Toggle transitivity as shown in the figure below will show all artifacts connected to `REQ-3` and their connnections to other artifacts. The end user can therefore use this information as a starting point for performing impact analysis.

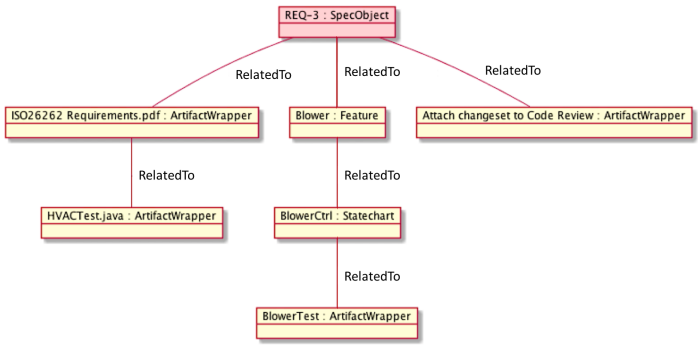


Figure 13: Capra graphical view showing transitively connected elements

Developer Guide

The following subsection describes the technical architecture of the tool. This information is also available in more detail in a tool demonstration paper (1). Our motivation for choosing this architecture design is based on a study on factors and guidelines that affect how a traceability tool can support traceability maintenance (2).

Architecture of Capra

Capra is an Eclipse plugin and uses the Eclipse Modeling Framework (EMF) as its base technology. It stores the traceability model as an EMF model. The tool relies on the Eclipse Extension mechanism and provides extension points for those parts of the tool that can be customized. Based on requirements we collected from many interested parties in the industry, the tool is customizable at four points:

1. The types of links to be supported;
2. Which types of artifacts can be traced to;
3. How the links should be stored;
4. The artifact handler that should be used in case there is more than one available for one artifact type.

Additionally, Capra has an API which makes traceability data available to other tools. The current version uses the provided traceability data to visualize it graphically.

The figure below depicts the extension points. The rationale for each of them is described in the following.

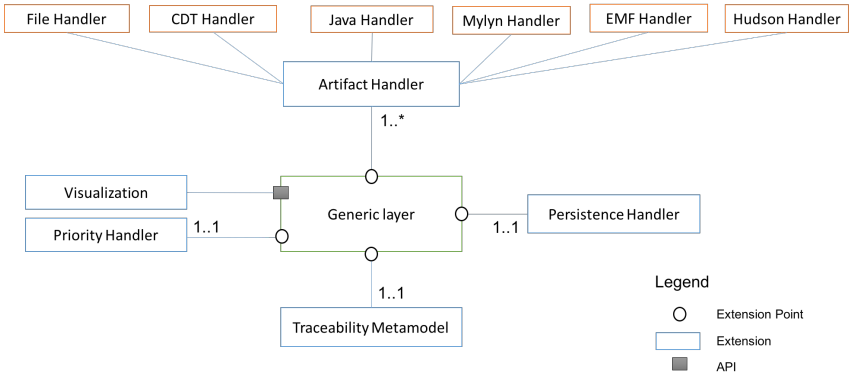


Figure 14: The Architecture of Capra

Traceability Metamodel

Depending on the company, development context, and process used, the traceability links required can differ. For example, traceability links for a company developing web-based solutions are not the same as links for companies developing embedded software. In the former case, traceability links can help connect certain entries in the server configuration files to specific requirements. The traceability links for embedded software need to relate, e.g., the hardware specification to the software design. Both concepts do not make sense in the respective other domain.

To address different link types, the tool offers an extension point for the traceability metamodel. Here the end user (company), can define the types of links through a metamodel and supply it to the tool. Examples of link types are “verifies”, “implements”, “refines”, “related to” etc. In addition to link types, the metamodel can also define additional information to be stored with each link. It might be desirable, e.g., to store the date and time the link was created or which user created it.

Artifact Handler

Software development usually involves a number of activities such as requirements engineering, design, implementation, and testing. In most cases, each of these activities use different tools and produce artifacts of different formats. A traceability tool needs to ensure that the different formats can be traced to and from. Since different companies use different tools, it is not easy to foresee which formats a traceability tool should support. This problem of diverse artifacts existing in the development environment has been noted by several studies on traceability. Our tool offers an extension point for Artifact Handlers which allows adding artifact formats based on the needs of the end users.

As discussed, Capra stores the traceability links as an EMF model. To be able to support tracing to other formats, EMF representations of these other formats are required. Implementing an extension for a certain format means providing an EMF representation of that format to the tool using the artifact handler extension point.

Persistence Handler

The storage of traceability links is another factor that can vary depending on company policies or project set-ups. For some cases it makes sense that there is a traceability model per project while in some cases there can be one traceability model for the whole workspace. The extension point Persistence Handler allows defining such storage locations. It will also allow integrating the traceability model with versioning solutions such as EMF Store, CDO or Git.

Priority Handler

In situations where there is more than one artifact handler that can handle the same artifact type, the tool provides an extension point for a so called Priority Handler. Here the user can define which handler should be used.

Capra API

Capra provides several programming interfaces that can be used by other plugins to access the traceability data. Currently, there are three interfaces: `ArtifactMetamodelAdapter`, `TraceMetamodelAdapter` and `TracePersistenceAdapter`. `ArtifactMetaodelAdapter` has methods that provide access to the artifact wrappers and their contents, `TraceMetamodelAdapter` has methods that provide access to the traceability links and the content of the links and the `TracePersistenceAdapter` has methods that provide access to the traceability model and the artifact wrapper model. The traceability model containing the traceability links is available to other tools. That means that traceability data can be used by other tools for specialised tasks such as impact analysis.

A good example on how these methods can be used is in the plugin `org.eclipse.capra.ui.plantuml`. This plugin utilizes the methods to get the traceability model and its links and also to determine which artifacts are connected by the links. The plugin uses the results of these methods to create a string that can be rendered as a diagram using the PlantUML view. For example in the file `VisualizationHelper`, the method `CreateMatrix()` calls a method `isThereATraceBetween()` which is part of the `TraceMetamodelAdapter` interface.

```
package org.eclipse.capra.ui.plantuml

import java.util.Collection

class VisualizationHelper {
    def static String createMatrix(EObject traceModel, Collection<EObject> firstElements, Collection<EObject> secondElements){
    val traceAdapter = ExtensionPointHelper.getTraceMetamodelAdapter().get()
    '''
    @startuml
    salt
    {#
    «IF firstElements != null»
    .«FOR e : secondElements»|«EMFHelper.getIdentifier(e)»«ENDFOR»
    «FOR first : firstElements»
    «EMFHelper.getIdentifier(first)»«FOR second : secondElements»|«IF traceAdapter.isThereATraceBetween(first, second, traceModel)
    «ENDFOR»
    «ELSE»
    Choose two containers to show a traceability matrix of their contents.
    «ENDIF»
    }
    @enduml
    '''
    }
```

Figure 15: Use of the method “isThereATraceBetween()” provided by the `TraceMetamodelAdapter` interface

Adding a custom Traceability Metamodel

To define your own traceability metamodel follow the steps below:

1. Create a Java project and name it `org.eclipse.capra.MyTraceabilityMetaModel`
2. Create a new folder and name it `model`
3. In the `model` folder create a new file and name it `MyTraceabilityMetaModel.xcore`. A pop up window will appear asking if you want to add the Xtext nature to the project. Click “Yes”.
4. Define your traceability metamodel as required. In our example, we add two types of traceability links i.e “implements” and “tests”

```
package org.eclipse.capra.mytraceabilityMetamodel

import org.eclipse.emf.ecore.EObject

class MyTraceModel {
    contains TraceLinks [0..*] item
}

abstract class TraceLinks{
    refers EObject [0..*] item
}

class Implements extends TraceLinks{

}

class Tests extends TraceLinks{

}
```

Figure 16: Tracemetamodel Definition

5. Open the `plugin.xml` file of the new project and click on the “Extension Points” tab
6. Un-check the check box that says “Show only extension points from the required plugins”
7. In the text box for the “Extension point filter” type “Traceability” and select `org.eclipse.capra.configuration.TraceabilityMetaModel`
8. Click Finish.

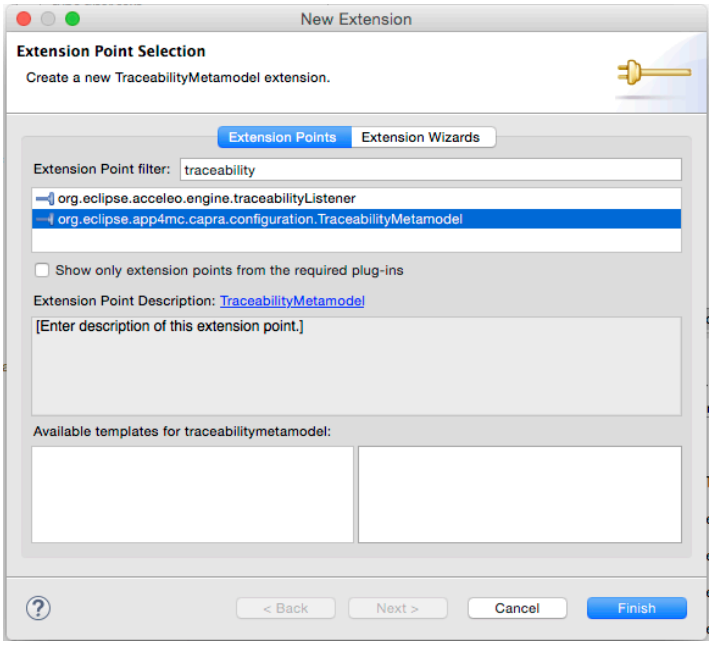


Figure 17: Adding the Trace metamodel extension point

9. Another pop up window will appear asking if you want to add the project `org.eclipse.capra.core` to the list of plugin dependencies. Click Yes
10. Right click on the newly added extension, and click on “New” then `TraceabilityMetaModelAdapter`. A new `TraceabilityMetaModelAdapter` will be created

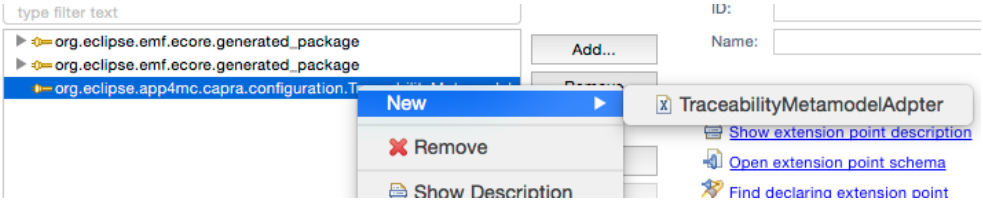


Figure 18: Adding the Trace metamodel extension point

11. Click on the newly created `TraceabilityMetaModelAdapter`. On the right hand side, we need to provide a class for this extension in which we will implement all the required interfaces.
12. Click on “Class” and a pop up window for creating a new class will appear. Make sure the folder is `src` and name your class

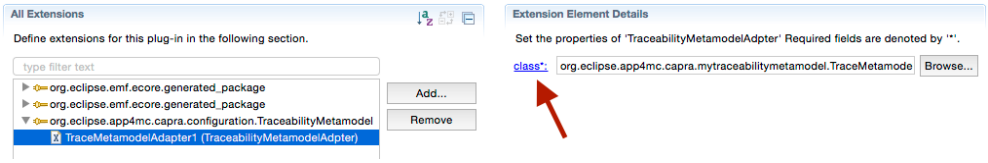


Figure 19: Add new Class

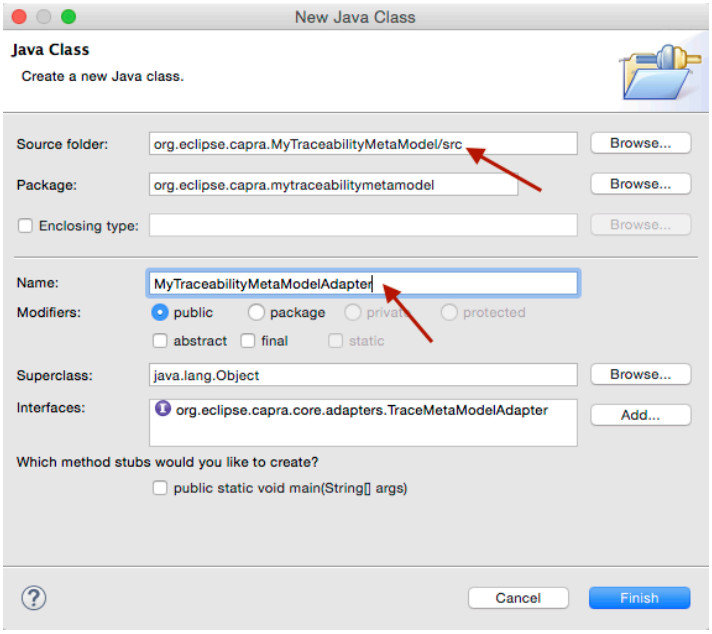


Figure 20: Add new Trace Adapter Class

13. Click Finish and a class will be created with the methods to be implemented in it.
14. Implement all the methods according to your new custom metamodel. Note that information about what each method does and its parameters can be obtained by hovering the mouse over the respective method name.

Adding a custom Artifact metamodel

To define your own artifact metamodel, follow the steps below:

1. In the same project used to define the traceability metamodel, go to the `model` folder, create a new file and name it `artifact.xcore`.
2. Define the artifact metamodel as in the picture below. Note that you can modify this definition to fit your needs.

```
1  package org.eclipse.capra.MyAtrifactMetaModel
2
3  class ArtifactWrapperContainer{
4      contains ArtifactWrapper [0..*] artifacts
5  }
6
7  class ArtifactWrapper {
8      String uri
9      String name
10     String ArtifactHandler
11 }
```

Figure 21: Artifact Metamodel definition

3. Next, open the `plugin.xml` file of the project and click on the “Extension Points” tab
4. Un-check the checkbox that says “Show only extension points from the required plugins”
5. In the textbox for “Extension point filter” type “Artifact” and select `org.eclipse.capra.configuration.ArtifactMetamodel`
6. Right click on the newly added extension, and click on “New” then `ArtifactMetamodelAdapter`. A new `ArtifactMetamodelAdapter` will be created

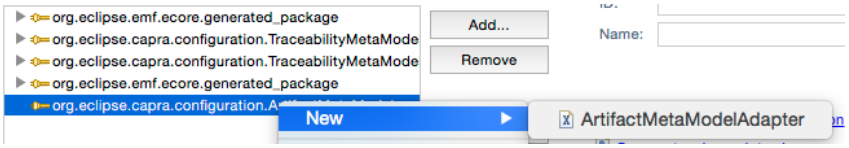


Figure 22: Artifact adapter definition

7. Click on “Class” and a pop up window for creating a new class will appear. Make sure the folder is `src` and name your class

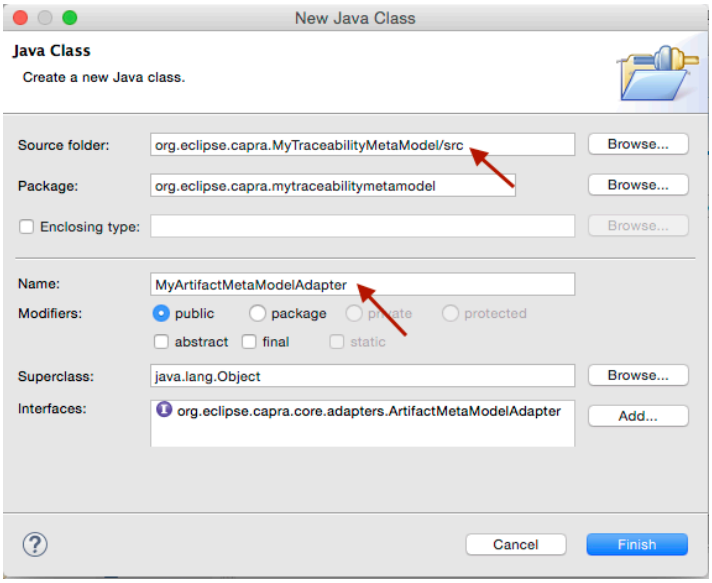


Figure 23: Artifact adapter Class definition

8. Click Finish and a class will be created with the methods to be implemented in it.
9. Implement all the methods according to your new custom metamodel. Note that information about what each method does and its parameters can be obtained by hovering the mouse over the respective method name.

NOTE: To test your new Traceability metamodel and artifact model, first close the project `org.eclipse.capra.generic.tracemodels`. Otherwise that project will be used by Capra.

Adding a new artifact handler

In case you want Capra to support an artifact type that is not already supported, you will need to create a new artifact handler for the particular artifact type.

As an example, we describe how the Java artifact handler was added using the following steps:

1. Create a new plugin project and name it `org.eclipse.capra.handler.jdt`.
2. In the `src` folder of the new project, create a package and name it `org.eclipse.capra.handler.jdt`.
3. Expand the `META-INF` folder , open the `MANIFEST.MF` file and click on the “Extensions” tab
4. Click on Add. A pop up window will appear with a list of available extension points.
5. Un-check the checkbox that says “Show only extension points from the required plugins”
6. In the textbox for “Extension point filter” type “Artifact” and select `org.eclipse.capra.configuration.ArtifactMetaModel` and click Finish.
7. Another pop up window will appear asking if you want to add the project `org.eclipse.capra.core` to the list of plugin dependencies. Click Yes
8. Right click on the newly added extension, and click on “New”, then “ArtifactHandler”. A new `ArtifactHandler` will be created.

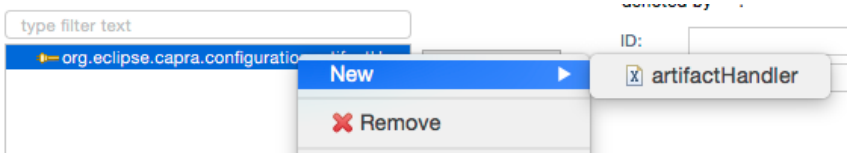


Figure 24: New Artifact Handler definition

9. Click on the newly created `ArtifactHandler`. On the right hand side, we need to provide a class for this extension, where we will implement all the required interfaces.
10. Click on “Class” and a pop up window for creating a new class will appear. Make sure the folder is `src` and name your class

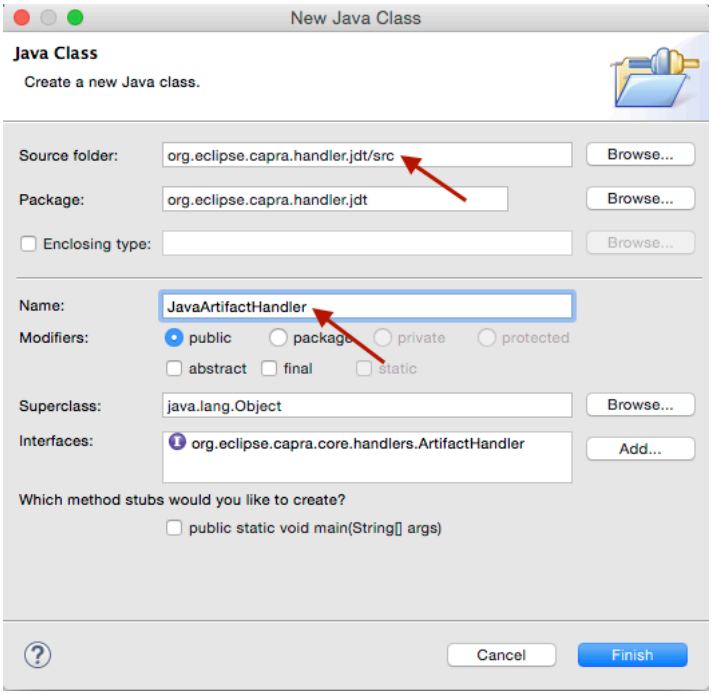


Figure 25: New Artifact Handler Class definition

11. Click Finish and a class will be created with the methods to be implemented in it. In this case there are only two methods.
12. Implement all required methods. Note that information about what each method does and its parameters can be obtained by hovering the mouse over the respective method name.

Changing the storage location of the traceability model

The storage of the traceability model and the artifact handler model is not fixed and can be modified depending on the users' needs and requirements. To change the storage location of the traceability model there are two options.

1. To edit the existing persistence handler project OR
2. To create a completely new persistence handler.

Edit existing Persistence Handler

3. Expand the project `org.eclipse.capra.generic.persistance`.
4. Expand the `src` folder and then the `org.eclipse.capra.generic.persistence` package
5. Open the file `TracePersistenceAdapter.java`.
6. Edit the static variables `DEFAULT_PROJECT_NAME`, `DEFAULT_TRACE_MODEL_NAME` and `DEFAULT_ARTIFACT_WRAPPER_MODEL_NAME` to reflect the new location and new model names for your traceability model and artifact wrapper model.

```
2⊕ * Copyright (c) 2016 Chalmers | University of Gothenburg, rt-labs and others.
11 package org.eclipse.capra.generic.persistance;
12
13⊕ import java.io.IOException;
29
30 public class TracePersistenceAdapter
31         implements org.eclipse.capra.core.adapters.TracePersistenceAdapter {
32
33     private static final String DEFAULT_PROJECT_NAME = "__WorkspaceTraceModels";
34     private static final String DEFAULT_TRACE_MODEL_NAME = "traceModel.xmi";
35     private static final String DEFAULT_ARTIFACT_WRAPPER_MODEL_NAME = "artifactWrappers.xmi";
```

Figure 26: Editing existing Persistence Handler

7. Save the project and Run Capra

Add a new Persistence Handler

8. Create a new plugin project and name it `org.eclipse.capra.MyPersistenceHandler`
9. In the `src` folder create a package and name it `org.eclipse.capra.MyPersistenceHandler`.
10. Expand the `META-INF` folder , Open the `MANIFEST.MF` file and click on the “Extensions” tab.
11. Click on Add. A pop up window will appear with a list of available extension points.
12. Un-check the checkbox that says “Show only extension points from the required plugins”
13. In the textbox for “Extension point filter” type “Persistence”, select `org.eclipse.capra.configuration.persistenceHandler` and

click Finish.

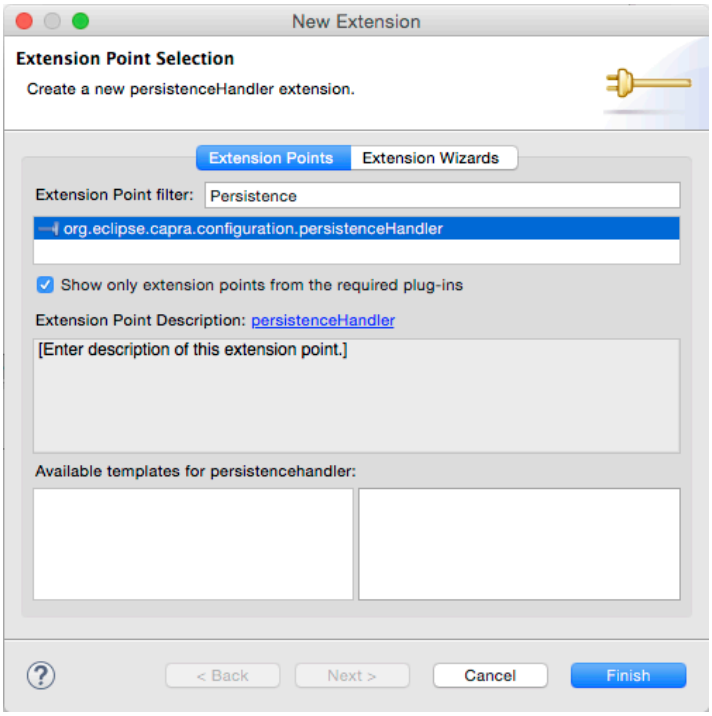


Figure 27: Add new Persistence Handler

14. Another pop up window will appear asking if you want to add the project `org.eclipse.capra.core` to the list of plugin dependencies. Click Yes.
15. Right click on the newly added extension, and click on “New”, then `persistenceHandler`. A new Persistence Handler will be created.

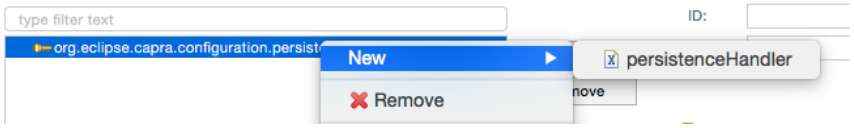


Figure 28: New Persistence Handler

16. Click on the newly created `persistenceHandler`. On the right hand side, we need to provide a class for this extension, where we will implement all the required interfaces.
17. Click on “Class” and a pop up window for creating a new class will appear. Make sure the folder is `src` and name your class.

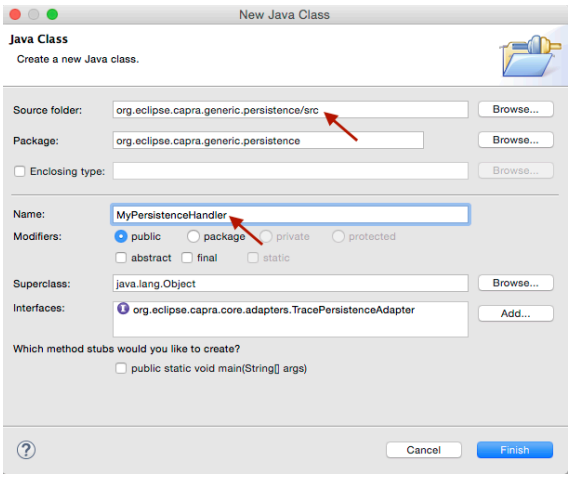


Figure 29: New Persistence Handler Class

18. Click Finish and a class will be created with the methods to be implemented in it. In this case there are only two methods.
19. Implement all methods according to your needs. Note that information about what each method does and its parameters can be obtained by hovering the mouse over the respective method name.

Taking care of multiple handlers for the same artifact type

There are cases in which several handlers are available for one artifact type. It is important during configuration to select which handler should be given a priority for the particular artifact type. This can be done by editing the code in the Priority Handler as follows:

1. Expand the project `org.eclipse.capra.generic.priority`.
2. Expand the src folder and then the `org.eclipse.capra.generic.priority` package.
3. Open the file `DefaultPriorityHander.java`.
4. Modify the code to select the correct handler. For instance, the code below selects a `hudsonHandler` when the element selected is a Test element or a build element.

```
 * Copyright (c) 2016 Chalmers | University of Gothenburg, rt-labs and others.
package org.eclipse.capra.generic.priority;

import java.util.Collection;

public class DefaultPriorityHandler implements PriorityHandler{

    @Override
    public ArtifactHandler getSelectedHandler(Collection<ArtifactHandler> handlers, Object selectedElement) {
        if (selectedElement instanceof TestElement || selectedElement instanceof BuildElement)
        {
            return handlers.stream().filter(h -> h instanceof HudsonHandler).findAny().get();

        }
        return null;
    }

}
```

Figure 30: Code for the Priority Handler

Adding new source files

It is important to maintain the correct copyright messages, indicating the contributors of each file and that it is covered by the EPL. You can use automation to insert a correct copyright header.

Install the Eclipse Releng Tools. They contain the copyright tool. Use the following copyright header:

```
Copyright (c) ${date} Chalmers | University of Gothenburg, rt-labs and others.
All rights reserved. This program and the accompanying materials
are made available under the terms of the Eclipse Public License v1.0
which accompanies this distribution, and is available at
http://www.eclipse.org/legal/epl-v10.html

Contributors:
     Chalmers | University of Gothenburg and rt-labs - initial API and implementation and/or initial documentation
```

The Contributors entry can be replaced with the appropriate names. Use “Fix copyrights” from the context menu to add the copyrights to all relevant files in a project or folder.

Rerefences

1. Maro, S. and Steghöfer, JP., 2016, September. Capra: A Configurable and Extendable Traceability Management Tool. In 2016 IEEE 24th International Requirements Engineering Conference (RE). IEEE.
2. Maro, S., Anjorin A., Wohlrab R. and Steghöfer, JP., 2016, September. Traceability Maintenance: Factors and Guidelines. In Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference. IEEE.