CobiGen — Code-based incremental Generator

2017-01-30

Table of Contents

Document Description

This document contains the documentation of the CobiGen core module as well as all CobiGen plug-ins and the CobiGen eclipse integration.

Current versions:

- CobiGen - Eclipse Plug-in v2.0.1
- CobiGen - Maven Build Plug-in v2.0.1

- CobiGen v3.0.0
- Cobigen - Java Plug-in v1.4.0
- CobiGen - XML Plug-in v3.0.0
- CobiGen - Property Plug-in v1.1.1
- CobiGen - Text Merger v1.1.0
- CobiGen - Sencha Plug-in v1.0.0
- CobiGen - JSON Plug-in v1.0.0

Authors:

- Malte Brunnlieb <malte.brunnlieb@capgemini.com>
- Steffen Holzer <steffen.holzer@capgemini.com>
- Ruben Diaz Martinez <ruben.diaz-martinez@capgemini.com>
- Fabian Kreis <fabian.kreis@capgemini.com>
- Lukas G??rlach <lukas.goerlach@capgemini.com>
- Marco Rose <marco.rose@capgemini.com>

1 Guide to the Reader

Dependent on the intention you are reading this document, you might be most interested in the following chapters:

- If this is **your first contact with CobiGen**, you will be interested in the general purpose of CobiGen, in the licensing of CobiGen, as well as in the Shared Service provided for CobiGen. Additionally, there are some general use cases, which are currently implemented and maintained to be used out of the box.
- As a **user of the CobiGen Eclipse integration**, you should focus on the Installation and Usage chapters to get a good introduction how to use CobiGen in eclipse.
- As a **user of the Maven integration**, you should focus on the Maven configuration chapter, which guides you through the integration if CobiGen into your build configuration.

- If you like to **adapt the configuration of CobiGen**, you have to step more deeper into the configuration guide as well as into the plug-in configuration extensions for the Java Plug-in, XML-Plugin, Java Property Plug-in, as well as for the Text-Merger Plug-in.
- Finally, if want to **develop your own templates**, you will be thankful for helpful links in addition to the plug-ins documentation as referenced in the previous point.

1. CobiGen - Code-based incremental Generator

1.1 Overview

CobiGen is a **generic incremental generator** for end to end code generation tasks, mostly used in Java projects. Due to a template-based approach, CobiGen **generates any set of text-based documents and document fragments**.

Input:

- Java classes
- Possibly more inputs like wsdl, which is currently not implemented.

Output:

- any text-based document or document fragments specified by templates

[[home_features-and characteristics]] == Features and Characteristics

- Generate fresh files across all the layers of a application - ready to run.
- Add on to existing files merging code into it. E.g. generate new methods into existing java classes or adding nodes to an XML file. Merging of contents into existing files will be done using structural merge mechanisms.
- Structural merge mechanisms are currently implemented for Java, XML, Java Property Syntax.
- Conflicts can be resolved individually but automatically by former configuration for each template.
- CobiGen provides an Eclipse integration as well as a Maven Integration.
- CobiGen comes with an extensive documentation for users and developers.
- templates can be fully tailored to project needs - this is considered as a simple task.

[[home_selection-of current and past cobigen applications]] == Selection of current and past CobiGen applications

General applications:

- Generation of a **Java CRUD application based on OASP architecture** including all software-layers on the server plus code for js-clients (AngularJs + SenchaJs). You can find details here.
- Generation of a **Java CRUD application according to the Register Factory architecture**. Persistence entities are the input for generation.
- Generation of **builder classes for generating testdata** for JUnit-Tests. Input are the persistence entities.

Project-specific applications in the past:

- Generation of an **additional Java type hierarchy on top of existing Java classes** in combination with additional methods to be integrated in the modified classes. Hibernate entites were considered as input as well as output of the generation. The rational in this case, was to generate an additional business object hierarchy on top of an existing data model for efficient business processing.

- Generation of **hash- and equals-methods** as well as copy constructors dependending on the field types of the input Java class. Furthermore, CobiGen is able to re-generate these methods/ constructors triggered by the user, i.e, when fields have been changed.
- **Extraction of JavaDoc** of test classes and their methods for generating a csv test documentation. This test documentation has been further processed manually in Excel to provide an good overview about the currently available tests in the software system, which enables further human analysis.

2. License Agreement

The usage of this software is subject to the terms and conditions of the Capgemini CobiGen License Agreement. To the extent that CobiGen contains software licensed under open source licenses, the respective license terms apply. The rights for using the open source software are granted directly by the respective owner of those rights as set forth in the corresponding open source license terms. To the extent CobiGen contains Capgemini proprietary code, non-transferrable use rights are granted by Capgemini as set forth in the CobiGen License Agreement.

[[mgmt_license-agreement_capgemini-internal usage]] == Capgemini internal usage CobiGen is freely usable for all employees of the Capgemini Deutschland GmbH. This includes working students, who have concluded a contract with the Capgemini Deutschland GmbH.

[[mgmt_license-agreement_distribution-to third parties]] == Distribution to third parties Distributing CobiGen to third parties, also includes employees of different companies of the Capgemini group, could only be done by concluding a software license contract for CobiGen with the third party.

Please contact me (Malte Brunnlieb) for retrieving the Licence Agreement, if needed.

3. Shared Service

As a generic generation tool, CobiGen has been established as the basis for a new Shared Service in Germany.

Service offer: development of a project-specific generator at a fixed price

The shared service will cover the creation of the templates and the configuration of CobiGen to the projects needs. Furthermore, there might be the need of further development of CobiGen according to a projects requirements. This can be also covered by the shared service after a more detailed discussion of the requirements and weighing up, whether the extension will be integrable into a general CobiGen release.

3.1 Preconditions

For using the shared service you should meet the following preconditions:

- provision of the contents to be generated as an example to derive templates from
- provision of input data---real world inputs if possible---to generate the contents from

According to this information, we are able to investigate

- whether the intended generation tasks could be implemented with CobiGen
- whether we need to adapt the current CobiGen implementation
- the fixed price of the service offering

4. General use cases

In addition to the selection of CobiGen applications introduced before, this chapter provides a more detailed overview about the currently implemented and maintained general use cases. These can be used by any project following a supported reference architecture as e.g. the OASP or Register Factory.

[[cobigen-usecases_open-application standard platform]] == Open Application Standard Platform

With our templates for OASP, you can generate a whole CRUD application from a single Entity class. You save the effort for creating, DAOs, Transfer Objects, simple CRUD use cases with REST services and even the client application can be generated.

[[cobigen-usecases_-crud server application for oasp4j]] === CRUD server application for OASP4J

For the server, the required files for all architectural layers (Data access, logic, and service layer) can be created based on your Entity class. After the generation, you have CRUD functionality for the entity from bottom to top which can be accessed via a RESTful web service. Details are provided in the Devon wiki.

[[cobigen-usecases_crud-client application for oasp4js]] === CRUD client application for OASP4JS

Based on the REST services on the server, you can also generate an Angular client based on OASP4JS. With the help of Node.js, you have a working client application for displaying your entities within minutes!

[[cobigen-usecases_testdata-builder for oasp4j]] === Testdata Builder for OASP4J

Generating a builder pattern for POJOs to easily create test data in your tests. CobiGen is not only able to generate a plain builder pattern but rather builder, which follow a specific concept to minimize test data generation efforts in your unit tests. The following `Person` class as an example:

Person class.

```
public class Person {

    private String firstname;
    private String lastname;
    private int birthyear;
    @NotNull
    private Address address;

    @NotNull
    public String getFirstname() {
        return this.firstname;
    }

    // additional default setter and getter
}
```

It is a simple POJO with a validation annotation, to indicate, that `firstname` should never be `null`. Creating this object in a test would imply to call every setter, which is kind of nasty. Therefore, the Builder Pattern has been introduced for quite a long time in software engineering, allowing to easily create POJOs with a fluent API. See below.

Builder pattern example.

```
Person person = new PersonBuilder()
                .firstname("Heinz")
                .lastname("Erhardt")
                .birthyear(1909)
                .address(
                    new AddressBuilder().postcode("22222")
                        .city("Hamburg").street("L??becker Str. 123")
                        .createNew())
                .addChild(
                    new PersonBuilder()[...].createNew()).createNew();
```

The Builder API generated by CobiGen allows you to set any setter accessible field of a POJO in a fluent way. But in addition lets assume a test, which should check the birth year as precondition for any business operation. So specifying all other fields of `Person`, especially `firstname` as it is mandatory to enter business code, would not make sense. The test behavior should just depend on the specification of the birth year and on no other data. So we would like to just provide this data to the test.

The Builder classes generated by CobiGen try to tackle this inconvenience by providing the ability to declare default values for any mandatory field due to validation or database constraints.

Builder Outline.

```
public class PersonBuilder {

    private void fillMandatoryFields() {
        firstname("lasdjfa??skdlfja");
        address(new AddressBuilder().createNew());
    };
    private void fillMandatoryFields_custom() {...};

    public PersonBuilder firstname(String value);
    public PersonBuilder lastname(String value);
    ...

    public Person createNew();
    public Person persist(EntityManager em);
    public List<Person> persistAndDuplicate(EntityManager em, int count);
}
```

Looking at the plotted builder API generated by CobiGen, you will find two `private` methods. The method `fillMandatoryFields` will be generated by CobiGen and regenerated every time CobiGen generation will be triggered for the `Person` class. This method will set every automatically detected field with not `null` constraints to a default value. However, by implementing `fillMandatoryFields_custom` on your own, you can reset these values or even specify more default values for any other field of the object. Thus, running `new PersonBuilder().birthyear(1909).createNew();` will create a valid object of `Person`, which is already pre-filled such that it does not influence the test execution besides the fact that it circumvents database and validation issues.

This even holds for complex data structures as indicated by `address(new AddressBuilder().createNew());`. Due to the use of the `AddressBuilder` for setting the default value for the field `address`, also the default values for `Address` will be set automatically.

Finally, the builder API provides different methods to create new objects.

- `createNew()` just creates a new object from the builder specification and returns it.
- `persist(EntityManager)` will create a new object from the builder specification and persists it to the database.
- `persistAndDuplicate(EntityManager, int)` will create the given amount of objects form the builder specification and persists all of these. After the initial generation of each builder, you might want to adapt the method body as you will most probably not be able to persist more than one object with the

same field assignments to the database due to `unique` constraints. Thus, please see the generated comment in the method to adapt `unqiue` fields accordingly before persisting to the database.

[[cobigen-usecases_custom-builder for business needs]] ==== Custom Builder for Business Needs

CobiGen just generates basic builder for any POJO. However, for project needs you probably would like to have even more complex builders, which enable the easy generation of more complex test data which are encoded in a large object hierarchy. Therefore, the generated builders can just be seen as a tool to achieve this. You can define your own business driven builders in the same way as the generated builders, but explicitely focusing on your business needs. Just take this example as a demonstration of that idea:

```
University uni = new ComplexUniversityBuilder()
  .withStudents(200)
  .withProfessors(4)
  .withExternalStudent()
  .createNew();
```

E.g. the method `withExternalStudent()` might create a person, which is a student and is flagged to be an external student. Basing this implementation on the generated builders will even assure that you would benefit from any default values you have set before. In addition, you can even imagine any more complex builder methods setting values driven your reusable testing needs based on the specific business knowledge.

4.1 Register Factory

[[cobigen-usecases_-crud server application]] === CRUD server application

Generates a CRUD application with persistence entites as inputs. This includes DAOs, TOs, use cases, as well as a CRUD JSF user interface if needed.

[[cobigen-usecases_-testdata builder]] === Testdata Builder

Analogous to [Testdata Builder for OASP4J]()

4.1.1 Test documentation

Generate test documentation from test classes. The input are the doclet tags of several test classes, which e.g. can specify a description, a cross-reference, or a test target description. The result currently is a csv file, which lists all tests with the corresponding meta-information. Afterwards, this file might be styled and passed to the customer if needed and it will be up-to-date every time!

5. CobiGen

5.1 Configuration

CobiGen will be configured using a configuration folder containing a context configuration, multiple template folders with a templates configuration per template folder, and a number of templates in each template folder. See some examples here. Thus, a simple folder structure might look like this:

```
CobiGen_Templates
 |- templateFolder1
    |- templates.xml
 |- templateFolder2
    |- templates.xml
 |- context.xml
```

[[cobigen-core_configuration_java-template logic]] == Java Template Logic

In addition, it is possible to implement more complex template logic by custom Java code *since cobigen-core-3.0.0 which is included in the Eclipse and Maven Plugin since version 2.0.0*. To enable this feature, you can simply turn the `CobiGen_Templates` into a simple maven project (if it is not already) and implement any Java logic in the common maven layout (e.g. in the source folder `src/main/java`). Each Java class will be instantiated by CobiGen for each generation process. Thus, you can even store any state within a Java class instance during generation. However, there is currently no guarantee according to the template processing order.

As a consequence, you have to implement your Java classes with a public default (non-parameter) constructor to be used by any template. Methods of the implemented Java classes can be called within templates by the simple standard FreeMarker expression for calling Bean methods: `SimpleType.methodName(param1)`. Until now, CobiGen will shadow multiple types with the same simple name indeterministically. So please prevent yourself from that situation.

Finally, if you would like to do some reflection within your Java code accessing any type of the template project or any type referenced by the input, you should load classes by making use of the classloader of the util classes. CobiGen will take care of the correct classloader building including the classpath of the input source as well as of the classpath of the template project. If you use any other classloader or build it by your own, there will be no guarantee, that generation succeeds.

5.1.1 Context Configuration

The context configuration (`context.xml`) always has the following root structure:

Context Configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<contextConfiguration xmlns="http://capgemini.com"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                      version="1.0">
    <triggers>
        ...
    </triggers>
</contextConfiguration>
```

The context configuration has a `version` attribute, which should match the XSD version the context configuration is an instance of. It should not state the version of the currently released version of CobiGen. This attribute should be maintained by the context configuration developers. If configured correctly, it will provide a better feedback for the user and thus higher user experience. Currently there is only the version v1.0. For further version there will be a changelog later on.

5.1.1.1 Trigger Node

As children of the `<triggers>` node you can define different triggers. By defining a `<trigger>` you declare a mapping between special inputs and a `templateFolder`, which contains all templates, which are worth to be generated with the given input.

trigger configuration.

```
<trigger id="..." type="..." templateFolder="..." inputCharset="UTF-8" >
    ...
</trigger>
```

- The attribute `id` should be unique within an context configuration. It is necessary for efficient internal processing.
- The attribute `type` declares a specific *trigger interpreter*, which might be provided by additional plug-ins. A *trigger interpreter* has to provide an *input reader*, which reads specific inputs and creates a template object model out of it to be processed by the FreeMarker template engine later on. Have a look at the plug-in's documentation of your interest and see, which trigger types and thus inputs are currently supported.
- The attribute `templateFolder` declares the relative path to the template folder, which will be used if the trigger gets activated.
- The attribute `inputCharset` *(optional)* determines the charset to be used for reading any input file.

5.1.1.2 Matcher Node

A trigger will be activated if its matchers hold the following formula:

`!(NOT || … || NOT) && AND && … && AND && (OR || … || OR)`

Whereas NOT/AND/OR describes the accumulationType of a *matcher* (see below) and e.g. `NOT` means 'a *matcher* with accumulationType NOT matches a given input'. Thus additionally to an *input reader*, a *trigger interpreter* has to define at least one set of *matchers*, which are satisfyable, to be fully functional. A `<matcher>` node declares a specific characteristics a valid input should have.

Matcher Configuration.

```
<matcher type="..." value="..." accumulationType="...">
    ...
</matcher>
```

- The attribute `type` declares a specific type of *matcher*, which has to be provided by the surrounding *trigger interpreter*. Have a look at the plug-in's documentation, which also provides the used trigger type for more information about valid matcher and their functionalities.
- The attribute `value` might contain any information necessary for processing the *matcher's* functionality. Have a look at the relevant plug-in's documentation for more detail.
- The attribute `accumulationType` *(optional)* specifies how the matcher will influence the trigger activation. Valid values are:

- OR (default): if any matcher of accumulation type OR *matches*, the trigger will be activated as long as there are no further matchers with different accumulation types
- AND: if any matcher with AND accumulation type does *not match*, the trigger will *not* be activated
- NOT: if any matcher with NOT accumulation type *matches*, the trigger will *not* be activated

5.1.1.3 VariableAssignment Node

Finally, a `<matcher>` node can have multiple `<variableAssignment>` nodes as children. *Variable assignments* allow to parametrize the generation by additional values, which will be added to the object model for template processing. The variables declared using *variable assignments*, will be made accessible in the templates.xml as well in the object model for template processing via the namespace `variables.*`.

Complete Configuration Pattern.

```
<?xml version="1.0" encoding="UTF-8"?>
<contextConfiguration xmlns="http://capgemini.com"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                      version="1.0">
    <triggers>
        <trigger id="..." type="..." templateFolder="...">
            <matcher type="..." value="...">
                <variableAssignment type="..." key="..." value="..." />
            </matcher>
        </trigger>
    </triggers>
</contextConfiguration>
```

- The attribute `type` declares the type of *variable assignment* to be processed by the *trigger interpreter* providing plug-in. This attribute enables *variable assignments* with different dynamic value resolutions.
- The attribute `key` declares the namespace under which the resolved value will be accessible later on.
- The attribute `value` might declare a constant value to be assigned or any hint for value resolution done by the *trigger interpreter* providing plug-in.

5.1.1.4 ContainerMatcher Node

The `<containerMatcher>` node is an additional matcher for matching containers of multiple input objects. Such a container might be a package, which encloses multiple types or---more generic---a model, which encloses multiple elements. A container matcher can be declared side by side with other matchers:

ContainerMatcher Declaration.

```
<?xml version="1.0" encoding="UTF-8"?>
<contextConfiguration xmlns="http://capgemini.com"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                      version="1.0">
    <triggers>
        <trigger id="..." type="..." templateFolder="..." >
            <containerMatcher type="..." value="..." retrieveObjectsRecursively="..." />
            <matcher type="..." value="...">
                <variableAssignment type="..." variable="..." value="..." />
            </matcher>
        </trigger>
    </triggers>
</contextConfiguration>
```

- The attribute `type` declares a specific type of *matcher*, which has to be provided by the surrounding *trigger interpreter*. Have a look at the plug-in's documentation, which also provides the used trigger type for more information about valid matcher and their functionalities.
- The attribute `value` might contain any information necessary for processing the *matcher's* functionality. Have a look at the relevant plug-in's documentation for more detail.
- The attribute `retrieveObjectsRecursively` *(optional boolean)* states, whether the children of the input should be retrieved recursively to find matching inputs for generation.

The semantics of a container matchers are the following:

- A `<containerMatcher>` does not declare any `<variableAssignment>` nodes
- A `<containerMatcher>` matches an input if and only if one of its enclosed elements satisfies a set of `<matcher>` nodes of the same `<trigger>`
- Inputs, which match a `<containerMatcher>` will cause a generation for each enclosed element

5.1.2 Templates Configuration

The template configuration (`templates.xml`) specifies, which templates exist and under which circumstances it will be generated. There are two possible configuration styles:

1. Configure the template meta-data for each template file by template nodes
2. *(since cobigen-core-v1.2.0)*: Configure templateScan nodes to automatically retrieve a default configuration for all files within a configured folder and possibly modify the automatically configured templates using templateExtension nodes

To get an intuition of the idea, the following will intially describe the first (more extensive) configuration style. Such an configuration root structure looks as follows:

Extensive Templates Configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<templatesConfiguration xmlns="http://capgemini.com"
                        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        version="1.0">
    <templates>
            ...
    </templates>
    <increments>
            ...
    </increments>
</templatesConfiguration>
```

The root node `<templatesConfiguration>` mainly specifies one attribute. The attribute `version` provides further usability support and will be handled analogous to the `version` attribute of the context configuration. The node `<templatesConfiguration>` allows two different grouping nodes as children. First, there is the `<templates>` node, which groups all declarations of templates. Second, there is the `<increments>` node, which groups all declarations about increments.

5.1.2.1 Template Node

The `<templates>` node groups multiple `<template>` declarations, which enables further generation. Each template file should be registered at least once as a template to be considered.

Example Template Configuration.

```
<templates>
    <template id="..." destinationPath="..." templateFile="..." mergeStrategy="..." targetCharset="..."
 />
    ...
</templates>
```

A template declaration consist of multiple information:

- The attribute `id` specifies an unique ID within the templates configuration, which will later be reused in the increment definitions.
- The attribute `destinationPath` specifies the destination path the template will be generated to. It is possible to use all variables defined by variable assignments within the path declaration using the FreeMarker syntax `${variables.*}`. While resolving the variable expressions, each dot within the value will be automatically replaced by a slash. This behavior is accounted for by the transformations of Java packages to paths as CobiGen has first been developed in the context of the Java world. Furthermore, the destination path variable resolution provides the following additional built-in operators analogue to the FreeMarker syntax:
  - `?cap_first` analogue to FreeMarker
  - `?uncap_first` analogue to FreeMarker
  - `?lower_case` analogue to FreeMarker
  - `?upper_case` analogue to FreeMarker
  - `?replace(regex, replacement)` - Replaces all occurrences of the regular expression `regex` in the variable's value with the given `replacement` string. (since cobigen-core v1.1.0)
  - `?removeSuffix(suffix)` - Removes the given `suffix` in the variable's value iff the variable's value ends with the given `suffix`. Otherwise nothing will happen. (since cobigen-core v1.1.0)
  - `?removePrefix(prefix)` - Analogue to `?removeSuffix` but removes the prefix of the variable's value. (since cobigen-core v1.1.0)
- The attribute `templateFile` describes the relative path dependent on the template folder specified in the trigger to the template file to be generated.
- The attribute `mergeStrategy` *(optional)* can be *optionally* specified and declares the type of merge mechanism to be used, when the `destinationPath` points to an already existing file. CobiGen by itself just comes with a `mergeStrategy override`, which enforces file regeneration in total. Additional available merge strategies have to be obtained from the different plug-in's documentations (see here for java, XML, properties, and text). Default: *not set* (means not mergable)
- The attribute `targetCharset` *(optional)* can be *optionally* specified and declares the encoding with which the contents will be written into the destination file. This also includes reading an existing file at the destination path for merging its contents with the newly generated ones. Default: *UTF-8*

5.1.2.2 TemplateScan Node

(since cobigen-core-v1.2.0)

The second configuration style for template meta-data is driven by initially scanning all available templates and automatically configure them with a default set of meta-data. A scanning configuration might look like this:

Example of Template-scan configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<templatesConfiguration xmlns="http://capgemini.com"
                        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        version="1.2">
    <templateScans>
        <templateScan templatePath="templates" templateNamePrefix="prefix_" destinationPath="src/main/
java"/>
    </templateScans>
</templatesConfiguration>
```

You can specify multiple `<templateScan …>` nodes for different `templatePaths` and different `templateNamePrefixes`.

- The `name` can be specified to later on reference the templates found by a template-scan within an increment. *(since cobigen-core-v2.1.)*
- The `templatePath` specifys the relative path from the `templates.xml` to the root folder from which the template scan should be performed.
- The `templateNamePrefix` *(optional)* defines a common id prefix, which will be added to all found and automatically configured templates.
- The `destinationPath` defines the root folder all found templates should be generated to, whereas the root folder will be a prefix for all found and automatically configured templates.

A `templateScan` will result in the following **default configuration of templates**. For each file found, new template will be created virtually with the following default values:

- `id`: file name without `.ftl` extension prefixed by `templateNamePrefix` from `template-scan`
- `destinationPath`: relative file path of the file found with the prefix defined by `destinationPath` from `template-scan`. Furthermore,
  - it is possible to use the syntax for accessing and modifying variables as described for the attribute `destinationPath` of the template node, besides the only difference, that due to file system restrictions you have to replace all ?-signs (for built-ins) with #-signs.
  - the files to be scanned, should provide their final file-ending by the following file naming convention: `<filename>.<fileending>.ftl` Thus the file-ending `.ftl` will be removed after generation.
- `templateFile`: relative path to the file found
- `mergeStrategy`: *(optional)* not set means not mergable
- `targetCharset`: *(optional)* defaults to UTF-8

5.1.2.3 TemplateExtension Node

(since cobigen-core-v1.2.0)

Additionally to the templateScan declaration it is easily possible to rewrite specific attributes for any scanned and automatically configured template.

Example Configuration of a TemplateExtension.

```
<templates>
    <templateExtension idref="prefix_FooClass.java" mergeStrategy="javamerge" />
</templates>

<templateScans>
    <templateScan templatePath="foo" templateNamePrefix="prefix_" destinationPath="src/main/java/foo"/>
</templateScans>
```

Lets assume, that the above example declares a `template-scan` for the folder `foo`, which contains a file `FooClass.java.ftl` in any folder depth. Thus the template scan will automatically create a virtual template declaration with `id=prefix_FooClass.java` and further default configuration.

Using the `templateExtension` declaration above will reference the scanned template by the attribute `idref` and overrides the `mergeStrategy` of the automatically configured template by the value `javamerge`. Thus we are able to minimize the needed templates configuration.

5.1.2.4 Increment Node

The `<increments>` node groups multiple `<increment>` nodes, which can be seen as a collection of templates to be generated. An increment will be defined by a unique `id` and a human readable `description`.

```
<increments>
    <increment id="..." description="...">
        <incrementRef idref="..." />
        <templateRef idref="..." />
        <templateScanRef ref="..." />
    </increment>
</increments>
```

An increment might contain multiple increments and/or templates, which will be referenced using `<incrementRef` …>, `<templateRef` …>, resp. `<templateScanRef` …> nodes. These nodes only declare the attribute `idref` or `ref` respectively, which will reference an increment, a template, or a template-scan by its `id` or `name`.

5.2 Plug-ins

5.2.1 CobiGen - Java Plug-in

The CobiGen Java Plug-in comes with a new input reader for java artifacts, new java related trigger and matchers, as well as a merging mechanism for Java sources.

5.2.1.1 Trigger extension

The Java Plug-in provides a new trigger for Java related inputs. It accepts different representations as inputs (see Java input reader) and provides additional matching and variable assignment mechanisms. The configuration in the `context.xml` for this trigger looks like this:

- type 'java'

  Example of a java trigger definition.

  ```
  <trigger id="..." type="java" templateFolder="...">
      ...
  </trigger>
  ```

  This trigger type enables Java elements as inputs.

Matcher types

With the trigger you might define matchers, which restrict the input upon specific aspects:

- type 'fqn' $\rightarrow$ full qualified name matching

  Example of a java trigger definition with a full qualified name matcher.

  ```
  <trigger id="..." type="java" templateFolder="...">
      <matcher type="fqn" value="(.+)\.persistence\.([^\.]+)\.entity\.([^\.]+)">
          ...
      </matcher>
  </trigger>
  ```

  This trigger will be enabled if the full qualified name (`fqn`) of the declaring input class matches the given regular expression (`value`).

- type 'package' $\rightarrow$ package name of the input

  Example of a java trigger definition with a package name matcher.

  ```
  <trigger id="..." type="java" templateFolder="...">
      <matcher type="package" value="(.+)\.persistence\.([^\.]+)\.entity">
          ...
      </matcher>
  </trigger>
  ```

  This trigger will be enabled if the package name (`package`) of the declaring input class matches the given regular expression (`value`).

- type 'expression'

  Example of a java trigger definition with a package name matcher.

  ```
  <trigger id="..." type="java" templateFolder="...">
      <matcher type="expression" value="instanceof java.lang.String">
          ...
      </matcher>
  </trigger>
  ```

  This trigger will be enabled if the expression evaluates to true. Valid expressions are

- `instanceof fqn`: checks an 'is a' relation of the input type
- `isAbstract`: checks, whether the input type is declared abstract

ContainerMatcher types

Additionally, the java plugin provides the ability to match packages (containers) as follows:

- type 'package'

  Example of a java trigger definition with a container matcher for packages.

  ```
  <trigger id="..." type="java" templateFolder="...">
      <containerMatcher type="package" value="com\.example\.app\.component1\.persistence.entity" />
  </trigger>
  ```

  The container matcher matches packages provided by the type `com.capgemini.cobigen.javaplugin.inputreader.to.PackageFolder` with a regular expression stated in the `value` attribute. (See containerMatcher semantics to get more information about containerMatchers itself.)

VariableAssignment types

Furthermore, it provides the ability to extract information from each input for further processing in the templates. The values assigned by variable assignments will be made available in template and the `destinationPath` of context.xml through the namespace `variables.<key>`. The Java Plug-in currently provides two different mechanisms:

- type 'regex' → regular expression group

```
<trigger id="..." type="java" templateFolder="...">
    <matcher type="fqn" value="(.+)\.persistence\.([^\.]+)\.entity\.([^\.]+)">
        <variableAssignment type="regex" key="rootPackage" value="1" />
        <variableAssignment type="regex" key="component" value="2" />
        <variableAssignment type="regex" key="pojoName" value="3" />
    </matcher>
</trigger>
```

This variable assignment assigns the value of the given regular expression group number to the given `key`.

- type 'constant' → constant parameter

```
<trigger id="..." type="java" templateFolder="...">
    <matcher type="fqn" value="(.+)\.persistence\.([^\.]+)\.entity\.([^\.]+)">
        <variableAssignment type="constant" key="domain" value="restaurant" />
    </matcher>
</trigger>
```

This variable assignment assigns the `value` to the `key` as a constant.

[[cobigen-javaplugin_java-input reader]] === Java input reader The Cobigen Java Plug-in implements an input reader for parsed java sources as well as for java `Class<?>` objects (loaded by reflection). So API user can pass `Class<?>` objects as well as `JavaClass` objects for generation. The latter depends on QDox, which will be used for parsing and merging java sources. For getting the right parsed java inputs you can easily use the `JavaParserUtil`, which provides static functionality to parse java files and get the appropriate `JavaClass` object.

Furthermore, due to restrictions on both inputs according to model building (see below), it is also possible to provide an array of length two as an input, which contains the `Class<?>` as well as the `JavaClass` object of the same class.

[[cobigen-javaplugin_template-object model]] ==== Template object model No matter whether you use reflection objects or parsed java classes as input, you will get the following object model for template creation:

- **pojo**
  - **name** ('String' :: Simple name of the input class)
  - **package** ('String' :: Package name of the input class)
  - **canonicalName** ('String' :: Full qualified name of the input class)
  - **annotations** ('Map<String, Object>' :: Annotations, which will be represented by a mapping of the full qualified type of an annotation to its value. To gain template compatibility, the key will

be stored with '_' instead of '.' in the full qualified annotation type. Furthermore, the annotation might be recursively defined and thus be accessed using the same type of mapping. Example `${pojo.annotations.javax_persistence_Id}`)

- **javaDoc** ('Map<String, String>') :: A generic way of addressing all available javaDoc doclets and comments. The only fixed variable is `comment` (see below). All other provided variables depend on the doclets found while parsing. The value of a doclet can be accessed by the doclets name (e.g. `${…javaDoc.author}`). doclets, which are declared multiple times, e.g., like `@param` are currently not handled sufficiently. By accessing you will get the value of the last occurrence.
  - **comment** ('String' :: javaDoc comment, which does not include any doclets)
- **extendedType** ('Map<String, Object>' :: The supertype, represented by a set of mappings *(since cobigen-javaplugin v1.1.0)*
  - **name** ('String' :: Simple name of the supertype)
  - **canonicalName** ('String' :: Full qualified name of the supertype)
  - **package** ('String' :: Package name of the supertype)
- **implementedTypes** ('List<Map<String, Object>>' :: A list of all implementedTypes (interfaces) represented by a set of mappings *(since cobigen-javaplugin v1.1.0)*
  - **interface** ('Map<String, Object>' :: List element)
    - **name** ('String' :: Simple name of the interface)
    - **canonicalName** ('String' :: Full qualified name of the interface)
    - **package** ('String' :: Package name of the interface)
- **fields** ('List<Map<String, Object>>' :: List of fields of the input class) *(renamed since cobigen-javaplugin v1.2.0; previously **attributes**)*
  - field ('Map<String, Object>' :: List element)
    - **name** ('String' :: Name of the Java field)
    - **type** ('String' :: Type of the Java field)
    - **canonicalType** ('String' :: Full qualified type declaration of the Java field’s type)
    - '**isId**' ('Deprecated' :: 'boolean' :: true if the Java field or its setter or its getter is annotated with the javax.persistence.Id annotation, false otherwise. Equivalent to `${pojo.attributes[i].annotations.javax_persistence_Id?has_content}`)
    - **javaDoc** (see pojo.javaDoc)
    - **annotations** (see pojo.annotations with the remark, that for fields all annotations of its setter and getter will also be collected)
- **methodAccessibleFields** ('List<Map<String, Object>>' :: List of fields of the input class or its inherited classes, which are accessible using setter and getter methods)

    - same as for *field* (but without javaDoc!)

  - **methods** ('List<Map<String, Object>>' :: The list of all methods, whereas one method will be represented by a set of property mappings)

    - method ('Map<String, Object>' :: List element)

      - **name** ('String' :: Name of the method)

      - **javaDoc** (see pojo.javaDoc)

      - **annotations** (see pojo.annotations)

Furthermore, when providing a `Class<?>` object as input, the Java Plug-in will provide additional functionalities as template methods:

1. `isAbstract(String fqn)` (Checks whether the type with the given full qualified name is an abstract class. Returns a boolean value.) *(since cobigen-javaplugin v1.1.1)*

2. `isSubtypeOf(String subType, String superType)` (Checks whether the `subType` declared by its full qualified name is a sub type of the `superType` declared by its full qualified name. Equals the Java expression `subType instanceof superType` and so also returns a boolean value.) *(since cobigen-javaplugin v1.1.1)*

Model Restrictions

As stated before both inputs (`Class<?>` objects and `JavaClass` objects) have their restrictions according to model building. In the following these restrictions are listed for both models, the ParsedJava Model which results from an `JavaClass` input and the RefelctedJava Model, which results from a Class<?>` input.

It is important to understand, that these restrictions are only present if you work with either Parsed Model **OR** the Reflected Model. If you use the *Maven Build Plug-in* or *Eclipse Plug-in* these two models are merged together so that they can mutually compensate their weaknesses.

Parsed Model

- annotations of the input???s supertype are not accessible due to restrictions in the QDox library. So `pojo.methodAccessibleFields[i].annotations` will always be empty for super type fields.

- annotations???s parameter values are available as Strings only (e.g. the Boolean value `true` is transformed into ???`true`???). This also holds for the Reflected Model.

- fields of ???supersupertypes??? of the input JavaClass are not available at all. So `pojo.methodAccessibleFields` will only contain the input type???s and the direct superclass???s fields.

- [resolved, since cobigen-javaplugin 1.3.1] field types of supertypes are always canonical. So `pojo.methodAccessibleFields[i].type` will always provide the same value as `pojo.methodAccessibleFields[i].canonicalType` (e.g. `java.lang.String` instead of the expected `String`) for super type fields.

Reflected Model

- annotations???s parameter values are available as Strings only (e.g. the Boolean value `true` is transformed into ???`true`???). This also holds for the Parsed Model.

- annotations are only available if the respective annotation has `@Retention(value=RUNTIME)`, otherwise the annotations are to be discarded by the compiler or by the VM at run time. For more information see RetentionPolicy.
- information about generic types is lost. E.g. a field???s/ methodAccessibleField???s type for `List<String>` can only be provided as `List<?>`.

5.2.1.2 Merger extensions

The Java Plug-in provides two additional merging strategies for Java sources, which can be configured in the `templates.xml`:

- Merge strategy `javamerge` (merges two Java resources and keeps the existing Java elements on conflicts)
- Merge strategy `javamerge_override` (merges two Java resources and overrides the existing Java elements on conflicts)

In general merging of two Java sources will be processed as follows:

Precondition of processing a merge of generated contents and existing ones is a common Java root class resp. surrounding class. If this is the case this class and all further inner classes will be merged recursively. Therefore, the following Java elements will be merged and conflicts will be resolved according to the configured merge strategy:

- `extends` and `implements` relations of a class: Conflicts can only occur for the extends relation.
- Annotations of a class: Conflicted if an annotation declaration already exists.
- Fields of a class: Conflicted if there is already a field with the same name in the existing sources. (Will be replaced / ignored in total, also including annotations)
- Methods of a class: Conflicted if there is already a method with the same signature in the existing sources. (Will be replaced / ignored in total, also including annotations)

5.2.2 Property Plug-in

The CobiGen Property Plug-in currently only provides different merge mechanisms for documents written in Java property syntax.

5.2.2.1 Merger extensions

There are two merge strategies for Java properties, which can be configured in the templates.xml:

- Merge strategy `propertymerge` (merges two properties documents and keeps the existing properties on conflicts)
- Merge strategy `propertymerge_override` (merges two properties documents and overrides the existing properties on conflicts)

Both documents (base and patch) will be parsed using the Java 7 API and will be compared according their keys. Conflicts will occur if a key in the patch already exists in the base document.

5.2.3 CobiGen - XML Plug-in

The CobiGen XML Plug-in comes with an input reader for xml artifacts, xml related trigger and matchers and provides different merge mechanisms for XML result documents.

[[cobigen-xmlplugin_trigger-extension]] == Trigger extension *(since cobigen-xmlplugin v2.0.0)*

The XML Plug-in provides a trigger for xml related inputs. It accepts xml documents as input (see XML input reader) and provides additional matching and variable assignment mechanisms. The configuration in the `context.xml` for this trigger looks like this:

- type 'xml'

  Example of a xml trigger definition.

```
<trigger id="..." type="xml" templateFolder="...">
    ...
</trigger>
```

  This trigger type enables xml documents as inputs.

[[cobigen-xmlplugin_matcher-types]] === Matcher types With the trigger you might define matchers, which restrict the input upon specific aspects:

- type 'nodename' $\rightarrow$ document’s root name matching

  Example of a xml trigger definition with a nodename matcher.

```
<trigger id="..." type="xml" templateFolder="...">
    <matcher type="nodename" value="\D\w*">
        ...
    </matcher>
</trigger>
```

  This trigger will be enabled if the root name of the declaring input document matches the given regular expression (`value`).

[[cobigen-xmlplugin_variableassignment-types]] === VariableAssignment types Furthermore, it provides the ability to extract information from each input for further processing in the templates. The values assigned by variable assignments will be made available in template and the `destinationPath` of context.xml through the namespace `variables.<key>`. The XML Plug-in currently provides only one mechanism:

- type 'constant' $\rightarrow$ constant parameter

```
<trigger id="..." type="xml" templateFolder="...">
    <matcher type="nodename" value="\D\w*">
        <variableAssignment type="constant" key="domain" value="restaurant" />
    </matcher>
</trigger>
```

This variable assignment assigns the `value` to the `key` as a constant.

[[cobigen-xmlplugin_xml-input reader]] === XML input reader The Cobigen XML Plug-in implements an input reader for parsed xml documents. So API user can pass `org.w3c.dom.Document` objects for generation. For getting the right parsed xml inputs you can easily use the `xmlplugin.util.XmlUtil`, which provides static functionality to parse xml files or input streams and get the appropriate `Document` object.

[[cobigen-xmlplugin_template-object]] ==== Template object Due to the heterogeneous structure an xml document can have, the xml input reader does not always create exactly the same model structure (in contrast to the java input reader). For example the model’s depth differs strongly, according to it’s input document. To allow navigational access to the nodes, the model also depends on the document's element's node names. All child elements with unique names, are directly accessable via their names. In addition it is possible to iterate over all child elements with held of the child list `Children`. So it is also possible to access child elements with non unique names.

The XML input reader will create the following object model for template creation (`EXAMPLEROOT, EXAMPLENODE1, EXAMPLENODE2, EXAMPLEATTR1,`… are just used here as examples. Of course they will be replaced later by the actual node or attribute names):

- **~EXAMPLEROOT~** ('Map<String, Object>' :: common element structure)
  - **_nodeName_** ('String' :: Simple name of the root node)
  - **_text_** ('String' :: Concatenated text content (PCDATA) of the root node)
  - **TextNodes** ('List<String>' :: List of all the root's text node contents)
  - **_at_~EXAMPLEATTR1~** ('String' :: String representation of the attribute's value)
  - **_at_~EXAMPLEATTR2~** ('String' :: String representation of the attribute's value)
  - **_at_…**
  - **Attributes** ('List<Map<String, Object>>' :: List of the root's attributes
    - at ('Map<String, Object>' :: List element)
      - **_attName_** ('String' :: Name of the attribute)
      - **_attValue_** ('String' :: String representation of the attribute's value)
  - **Children** ('List<Map<String, Object>>' :: List of the root's child elements
    - child ('Map<String, Object>' :: List element)
      - …common element sub structure…
  - **~EXAMPLENODE1~** ('Map<String, Object>' :: One of the root's child nodes)
    - …common element structure…
  - **~EXAMPLENODE2~** ('Map<String, Object>' :: One of the root's child nodes)
    - …common element sub structure…
    - **~EXAMPLENODE21~** ('Map<String, Object>' :: One of the nodes's child nodes)
      - …common element structure…
    - **~EXAMPLENODE…~**
  - **~EXAMPLENODE…~**

In contrast to the java input reader, this xml input reader does currently not provide any additional template methods.

5.2.3.1 Merger extensions

There are two merge strategies for XML documents, which can be configured in the templates.xml:

- Merge strategy `xmlmerge` (merges two XML documents and keeps the existing XML elements on conflicts)
- Merge strategy `xmlmerge_override` (merges two XML documents and overrides the existing XML elements on conflicts)

As XML is a meta-language, it is not possible to compare all the different XML-based languages in a correct semanitcally way. Thus the current merge algorithm is based on some documents, which have been interesting to be merged in the past, like, spring bean specifications and xml specifications of named queries.

The current algorithms looks like this:

1. If the two elements declare an `id` attribute
   - If the two elements declare an attribute `id`
     - the value of the attribute `id` will be compared and result of the comparison
2. If the two element’s qualified name is "import"
   - If the two elements declare an attribute `resource`
     - the value of the attribute `resource` will be compared and declared as the result of the comparison
3. If the two element’s qualified name is "query", "property", or "define"
   - If the two elements declare an attribute `name`
     - the value of the attribute `name` will be compared and declared as the result of the comparison
4. If the two element’s qualified name is "transition"
   - If the two elements declare an attribute `on`
     - the value of the attribute `on` will be compared and declared as the result of the comparison
5. If the two element’s qualified name is "outputLabel" or "message"
   - If the two elements declare an attribute `for` and an attribute `value`
     - both values will be compared and declared as the result of the comparison
   - If the two elements only declare the attribute `for`
     - the value of the attribute `for` will be compared and declared as the result of the comparison
6. If the two element’s parents are equal
   - the value text value will be compared and result of the comparison
7. All other elements will only be compared by the precondition (equal node name)

5.2.4 Text Merger Plug-in

The Text Merger Plug-in enables merging result free text documents to existing free text documents. Therefore, the algorithms are also very rudimentary.

5.2.4.1 Merger extensions

There are currently two merge strategies:

- merge strategy `textmerge_append` (appends the text directly to the end of the existing document)
- merge strategy `textmerge_appendWithNewLine` (appends the text after adding a new line break to the existing document) *Remark:* empty patches will not result in appending a new line any more since v1.0.1

5.2.5 Sencha Merger Plug-in

The sencha Merger Plug-in enables merging result Ext JS files to existing ones. This plug-in is used for generte an Ext JS 6 client with all CRUD functionalities enabled. The plug-in also generates de i18n files just appending at the end of the word the `ES` or `EN` suffixes, to put into the developer knowledge that this words must been translated to the correspondent language.

5.2.5.1 Merger extensions

There are currently two merge strategies:

- merge strategy `senchamerge` (add the new code respecting the existing is case of conflict)
- merge strategy `senchamerge_override` (add the new code overwriting the existent in case of conflict)

5.2.6 JSON Plug-in

??n future, the JSON plugin should be able to also merge arbitrary JSON files. However, currently due to business needs, we just support a specific merge algorithm for sencha architect configuration files. Thus, CobiGen is able to generate sencha architect projects for client generation and potentially add new views to already existing sench archtiect projects.

5.2.6.1 Merger extensions

There are currently two merge strategies:

- merge strategy `sencharchmerge` (add the new code respecting the existing is case of conflict)
- merge strategy `sencharchmerge_override` (add the new code overwriting the existent in case of conflict)

[[cobigen_maven-build plug-in]] == Maven Build Plug-in :leveloffset: 2 :toc: toc::[]

Part I. Maven Build Integration

For maven integration of CobiGen you can include the following build plugin into your build:

Build integration of CobiGen.

```
<build>
    <plugins>
        <plugin>
            <groupId>com.capgemini</groupId>
            <artifactId>cobigen-maven-plugin</artifactId>
            <version>VERSION-YOU-LIKE</version>
            <executions>
                <execution>
                    <id>cobigen-generate</id>
                    <phase>site</phase>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Available goals

- `generate`: Generates contents configured by the standard non-compiled configuration folder. Thus generation can be controlled/configured due to an location URI of the configuration and tempalte or increment ids to be generated for a set of inputs.

Available phases are all phases, which already provide compiled sources such that CobiGen can perform reflection on it. Thus possible phases are for example package, site.

6. CobiGen Configuration

For generation using the CobiGen maven plug-in, the CobiGen configuration can be provided in two different styles:

1. By a `configurationFolder`, which should be available on the file system whenever you are running the generation. The value of `configurationFolder` should correspond to the maven file path syntax.

   Provide CobiGen configuration by configuration folder (file).

   ```
   <build>
       <plugins>
           <plugin>
               ...
               <configuration>
                   <configurationFolder>cobigen-templates</configurationFolder>
               </configuration>
               ...
           </plugin>
       </plugins>
   </build>
   ```

2. By maven dependency, whereas the maven dependency should stick on the same conventions as the configuration folder. This explicitly means that it should contain non-compiled resources as well as the `context.xml` on top-level.

   Provide CobiGen configuration by maven dependency (jar).

   ```
   <build>
       <plugins>
           <plugin>
               ...
               <dependencies>
                   <dependency>
                       <groupId>com.capgemini</groupId>
                       <artifactId>cobigen-templates-XYZ</artifactId>
                       <version>VERSION-YOU-LIKE</version>
                   </dependency>
               </dependencies>
               ...
           </plugin>
       </plugins>
   </build>
   ```

   We currently provide a generic deployed version of the templates on the oasp-nexus for Register Factory (`<artifactId>cobigen-templates-rf</artifactId>`) and for the OASP itself (`<artifactId>cobigen-templates-oasp</artifactId>`).

7. Build Configuration

Using the following configuration you will be able to customize your generation as follows:

- `<destinationRoot>` specifies the root directory the relative `destinationPath` of CobiGen templates configuration should depend on. *Default ${basedir}*
- `<inputPackage>` declares a package name to be used as input for batch generation. This refers directly to the CobiGen Java Plug-in container matchers of type package configuration.
- `<inputFile>` declares a file to be used as input. The CobiGen maven plug-in will try to parse this file to get an appropriate input to be interpreted by any CobiGen plug-in.
- `<increment>` specifies an increment ID to be generated.
- `<template>` specifies a template ID to be generated.
- `<forceOverride>` specifies an overriding behavior, which enables non-mergable resources to be completely rewritten by generated contents. For mergable resources this flag indicates, that conflicting fragments during merge will be replaced by generated content. *Default: false*

Example for a simple build configuration.

```
<build>
    <plugins>
        <plugin>
            ...
            <configuration>
                <destinationRoot>${basedir}</destinationRoot>
                <inputPackages>
                    <inputPackage>package.to.be.used.as.input</inputPackage>
                </inputPackages>
                <inputFiles>
                    <inputFile>path/to/file/to/be/used/as/input</inputFile>
                </inputFiles>
                <increments>
                    <increment>IncrementID</increment>
                </increments>
                <templates>
                    <template>TemplateID</template>
                </templates>
                <forceOverride>false</forceOverride>
            </configuration>
            ...
        </plugin>
    </plugins>
</build>
```

8. Eclipse Plug-in

8.1 Installation

Remark: CobiGen is preinstalled in the oasp/oasp4j-ide.

8.1.1 Preconditions

- Eclipse 4.x
- Java 7 Runtime (for starting eclipse with CobiGen). This is independent from the target version of your developed code.

8.1.2 Installation steps

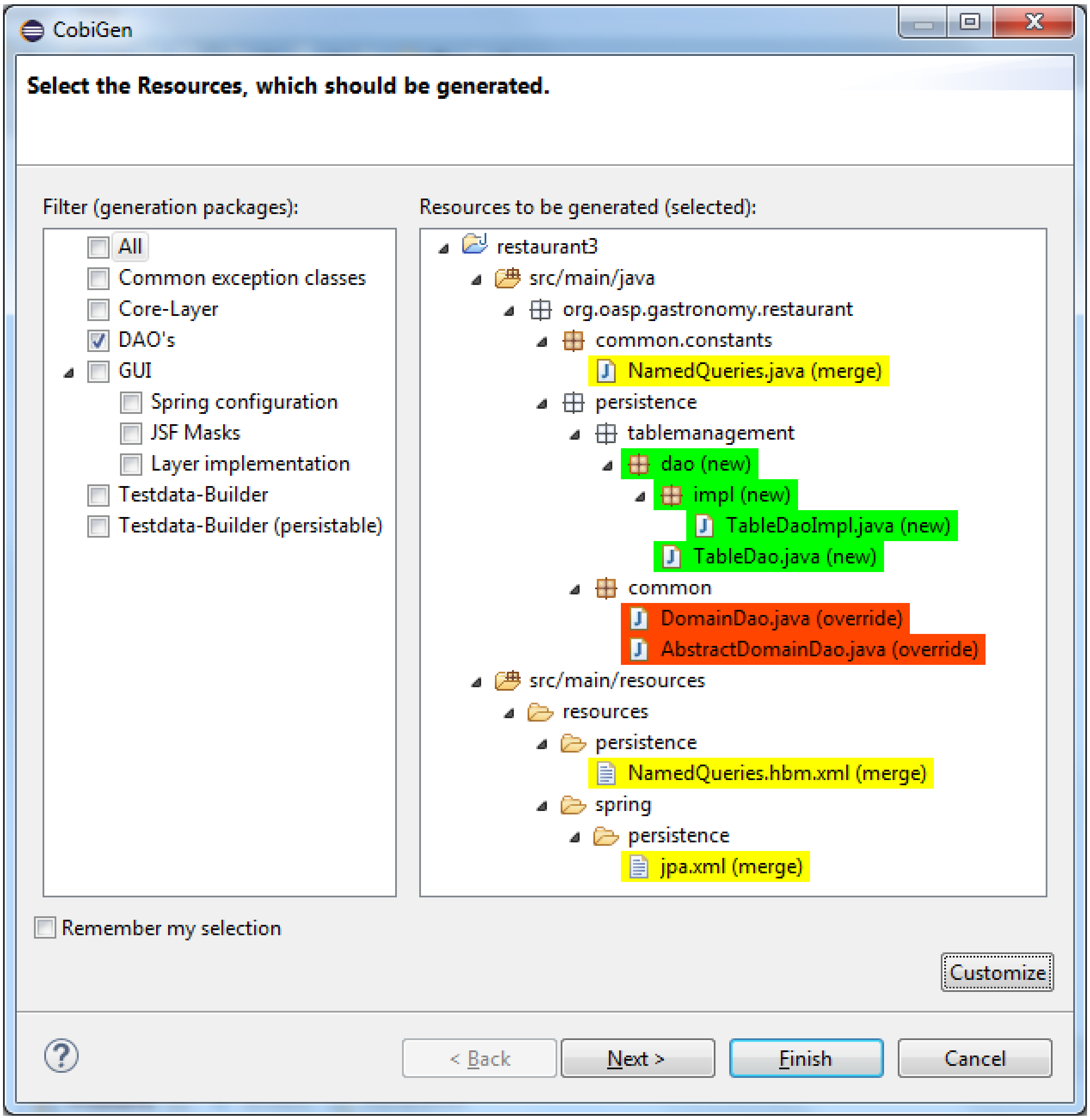
1. Open the eclipse installation dialog
   menu bar → *Help* → *Install new Software…*
2. Open CobiGen's updatesite as archive
   Add… → *Archive…* → Set *location* to "jar:file:////de-file10/ClientPE/OASP/CobiGenerator/Releases/stable/cobigen-eclipse.zip!/" → Choose an appropriate name → *OK*
3. Follow the installation wizard
   Select *CobiGen Eclipse Plug-in* → *Next* → *Next* → accept the license → *Finish* → *OK* → *Yes*
4. Once installed, a new menu entry named "CobiGen" will show up in the *Package Explorer's* context menu. In the sub menu. In the sub menu there will the *Generate…* command, which will start the generation wizard of CobiGen. For now, this may be greyed out.
5. Checkout (clone) your project's templates folder or use the current templates released with CobiGen (https://github.com/devonfw/tools-cobigen/tree/master/cobigen-templates) and then choose Import -> General -> Existing Projects into Workspace to import the templates into your workspace.
   Dependent on your context configuration menu entry *Generate…* may be greyed out or not. See context configuration chapter for more information about valid selections for generation.

8.2 Usage

CobiGen has two different generation modes depending on the input selected for generation. The first one is the *simple mode*, which will be started if the input contains only one input artifact, e.g. for Java an input artifact currently is a Java file. The second one is the *batch mode*, which will be started if the input contains multiple input artifacts, e.g. for Java this means a list of files. In general this means also that the batch mode might be started when selecting complex models as inputs, which contain multiple input artifacts. The latter scenario has only been covered in the research group,yet.

8.2.1 Simple Mode

Selecting the menu entry *Generate…* the generation wizard will be opened:



CobiGen

Select the Resources, which should be generated.

Filter (generation packages):

- [ ] All
- [ ] Common exception classes
- [ ] Core-Layer
- [✓] DAO's
- [ ] GUI
  - [ ] Spring configuration
  - [ ] JSF Masks
  - [ ] Layer implementation
- [ ] Testdata-Builder
- [ ] Testdata-Builder (persistable)

Resources to be generated (selected):

```
restaurant3
  src/main/java
    org.oasp.gastronomy.restaurant
      common.constants
        NamedQueries.java (merge)
      persistence
        tablemanagement
          dao (new)
            impl (new)
              TableDaoImpl.java (new)
            TableDao.java (new)
        common
          DomainDao.java (override)
          AbstractDomainDao.java (override)
  src/main/resources
    resources
      persistence
        NamedQueries.hbm.xml (merge)
      spring
        persistence
          jpa.xml (merge)
```

- [ ] Remember my selection

The left side of the wizard shows all available increments, which can be selected to be generated. Increments are a container like concept encompassing multiple files to be generated, which should result in a semantically closed generation output. On the right side of the wizard all files, which might be effected by the generation. Dependent on the increment selection on the left side, potenially effected files will be shown on the right side. The type of modification of each file will be encoded into following color scheme if the files are selected for generation:

- **green:** files, which are currently non-existent in the file system. These files will be created during generation
- **yellow:** files, which are currently existent in the file system and which are configured to be merged with generated contents.
- **red:** files, which are currently existent in the file system. These files will be overwritten if manually selected.

- **no color:** files, which are currently existent in the file system. Additionally files, which were unselected and thus will be ignored during generation.

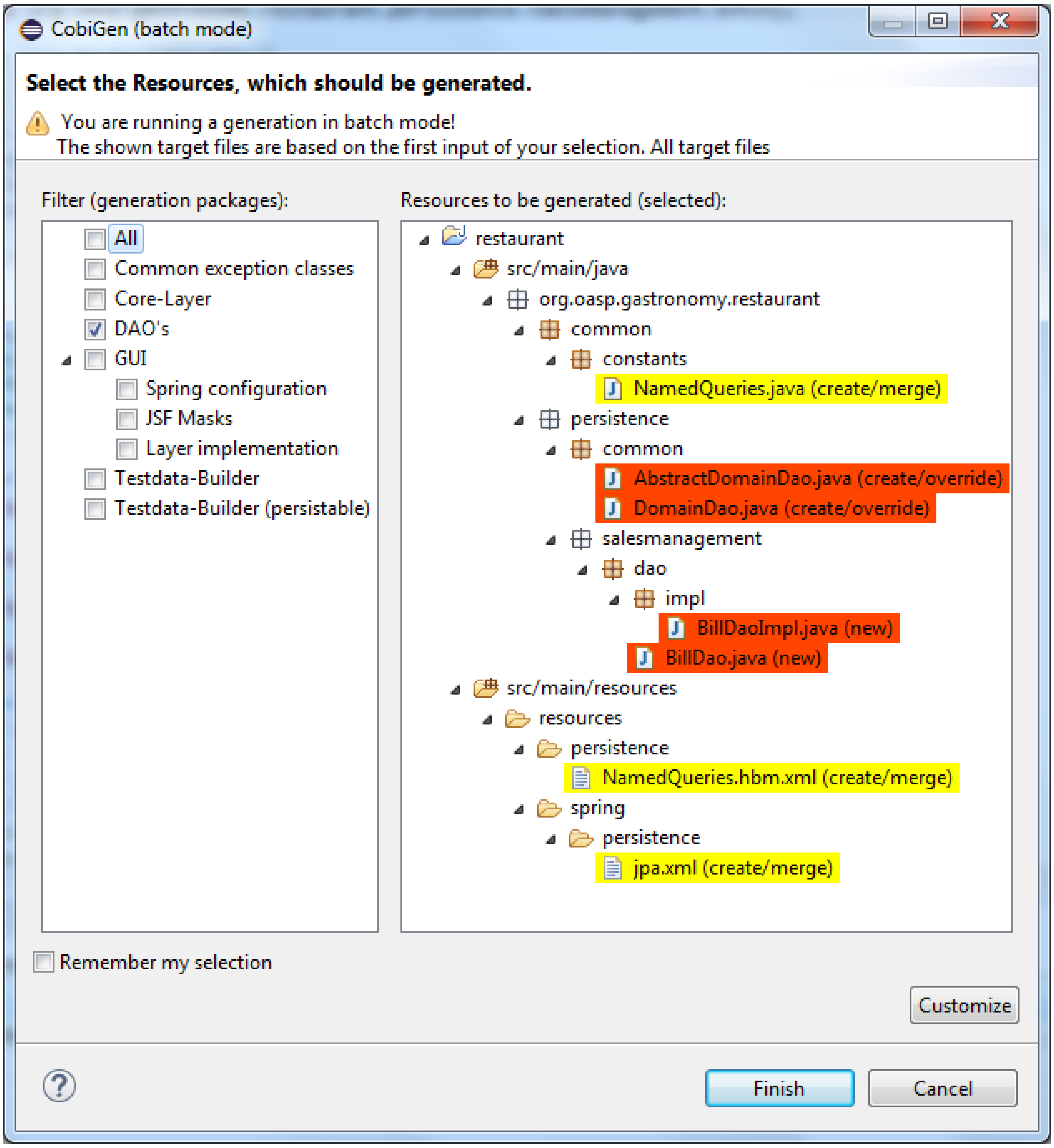
Selecting an increment on the left side will initialize the selection of all shown files to be generated on the right side, whereas green and yellow categorized files will be selected intially. A manual modification of the pre-selection can be performed by switching to the customization tree using the *Customize* button on the right lower corner.

> **Optional:** If you want to customize the generation object model of a Java input class, you might continue with the *Next >* button instead of finishing the generation wizard. The next generation wizard page is currently available for Java file inputs and lists all non-static fields of the input. Unselecting entries will lead to an adapted object model for generation, such that unselected fields will be removed in the object model for generation. By default all fields will be included in the object model.

Using the *Finish* button, the generation will be performed. Finally, CobiGen runs the eclipse internal *organize imports* and *format source code* for all generated sources and modified sources. Thus it is possible, that---especially *organize imports* opens a dialog if some types could not be determined automatically. This dialog can be easily closed by pressing on *Continue*. If the generation is finished, the *Success!* dialog will pop up.

8.2.2 Batch mode

Are there multiple input elements selected, e.g., Java files, CobiGen will be started in batch mode. For the generation wizard dialog this means, that the generation preview will be contrained to the first selected input element. It does *not* preview the generation for each element of the selection or of a complex input. The selection of the files to be generated will be generated for each input element analogously afterwards.


CobiGen (batch mode)
Select the Resources, which should be generated.
You are running a generation in batch mode!
The shown target files are based on the first input of your selection. All target files
Filter (generation packages):
All
Common exception classes
Core-Layer
DAO's
GUI
Spring configuration
JSF Masks
Layer implementation
Testdata-Builder
Testdata-Builder (persistable)
Resources to be generated (selected):
restaurant
src/main/java
org.oasp.gastronomy.restaurant
common
constants
NamedQueries.java (create/merge)
persistence
common
AbstractDomainDao.java (create/override)
DomainDao.java (create/override)
salesmanagement
dao
impl
BillDaoImpl.java (new)
BillDao.java (new)
src/main/resources
resources
persistence
NamedQueries.hbm.xml (create/merge)
spring
persistence
jpa.xml (create/merge)
Remember my selection
Customize
Finish
Cancel

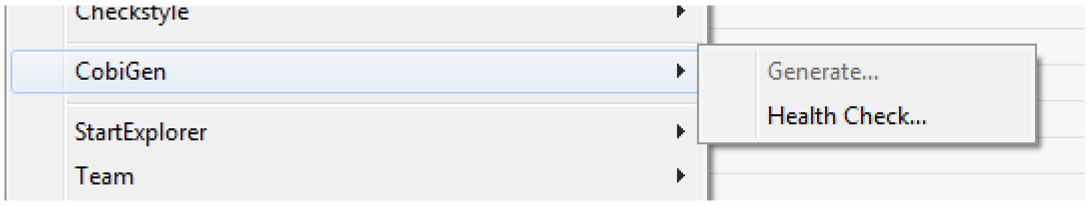
Thus the color encoding differs also a little bit:

- **yellow:** files, which are configured to be merged.
- **red:** files, which are not configured with any merge strategy and thus will be created if the file does not exist or overwritten if the file already exists
- **no color:** files, which will be ignored during generation

Initially all possible files to be generated will be selected.

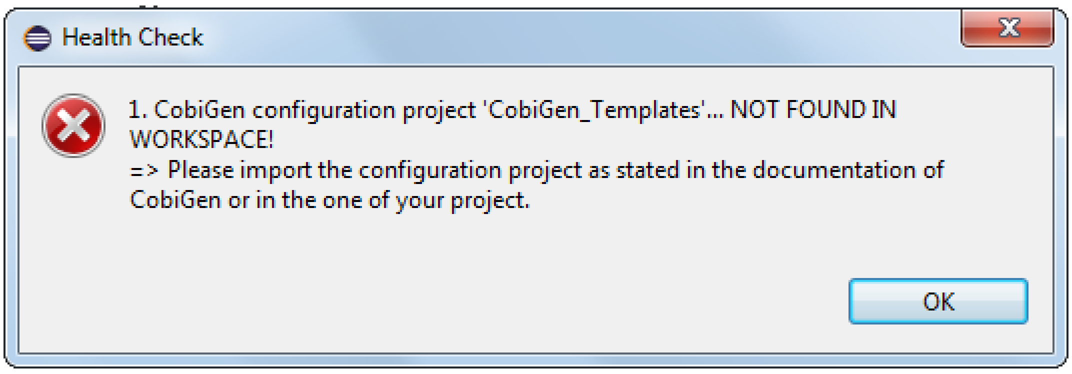
8.2.3 Health Check

To check whether CobiGen runs appropriately for the selected element(s) the user can perform a *Health Check* by activating the respective menu entry as shown below.



The simple *Health Check* includes 3 checks. As long as any of these steps fails, the *Generate* menu entry is grayed out.

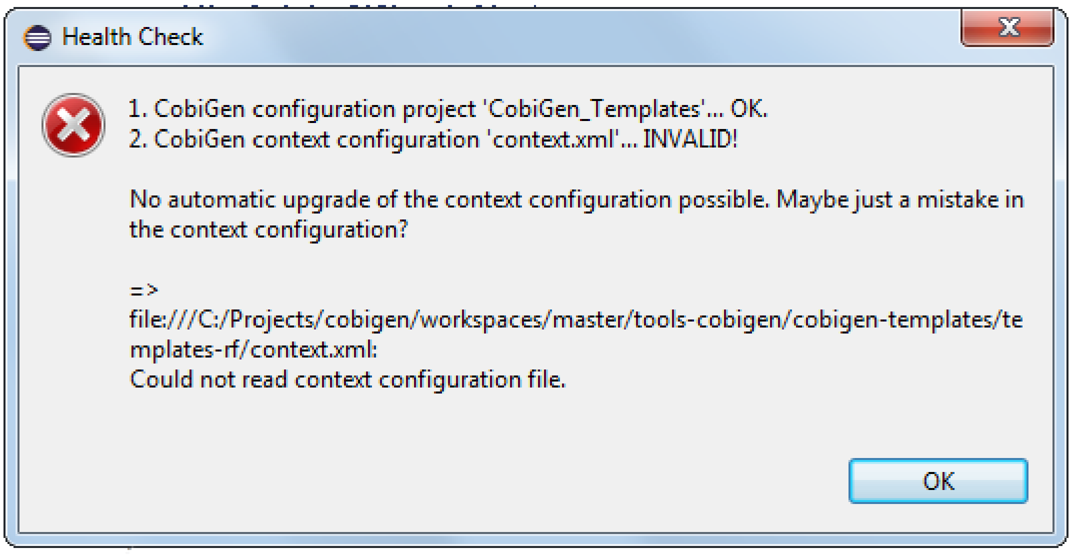
The first step is to check whether the generation configuration is available at all. If this check fails you will see the following message:



1. CobiGen configuration project 'CobiGen_Templates'... NOT FOUND IN WORKSPACE!
=> Please import the configuration project as stated in the documentation of CobiGen or in the one of your project.

This indicates, that there is no Project named *CobiGen_Templates* available in the current workspace. To run CobiGen appropriately, it is necessary to have a configuration project named *CobiGen_Templates* imported into your workspace. For more information see chapter Eclipse Installation.

The second step is to check whether the template project includes a valid *context.xml*. If this check fails, you will see the following message:



1. CobiGen configuration project 'CobiGen_Templates'... OK.
2. CobiGen context configuration 'context.xml'... INVALID!

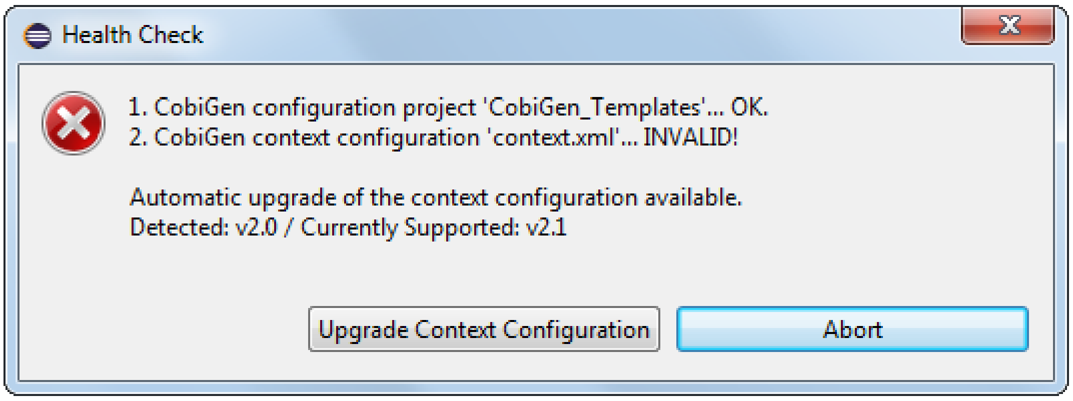
No automatic upgrade of the context configuration possible. Maybe just a mistake in the context configuration?

=>
file:///C:/Projects/cobigen/workspaces/master/tools-cobigen/cobigen-templates/templates-rf/context.xml:
Could not read context configuration file.

This means that either your *context.xml*

- does not exist (or has another name)
- or it is not valid one in any released version of CobiGen
- or there is simply no automatic routine of upgrading your context configuration to a valid state.

If all this is not the case, such as, there is a *context.xml*, which can be successfully read by CobiGen, you might get the following information:



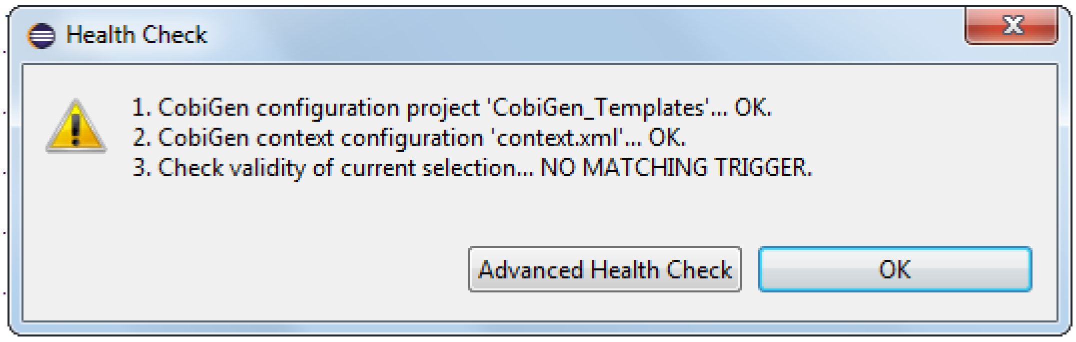
1. CobiGen configuration project 'CobiGen_Templates'... OK.
2. CobiGen context configuration 'context.xml'... INVALID!

Automatic upgrade of the context configuration available.
Detected: v2.0 / Currently Supported: v2.1

This means that your *context.xml* is available with the correct name but it is outdated (belongs to an older CobiGen version). In this case just click on *Upgrade Context Configuration* to get the latest version.

> **Remark:** This will create a backup of your current context configuration and converts your old configuration to the new format. The upgrade will remove all comments from the file, which could be retrieved later on again from the backup. If the creation of the backup fails, you will be asked to continue or to abort.

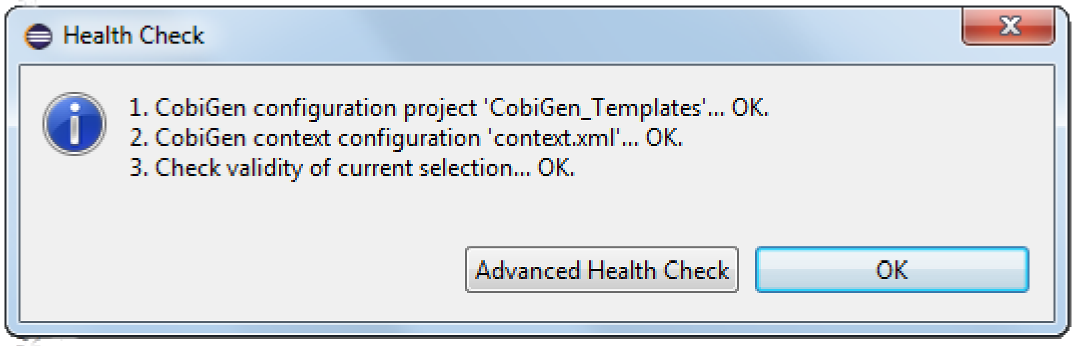
The third step checks whether there are templates for the selected element(s). If this check fails, you will see the following message:


Health Check

1. CobiGen configuration project 'CobiGen_Templates'... OK.
2. CobiGen context configuration 'context.xml'... OK.
3. Check validity of current selection... NO MATCHING TRIGGER.

This indicates, that there no trigger has been activated, which matches the current selection. The reason might be that your selection is faulty or that you imported the wrong template project (e.g. you are working on a OASP project, but imported the Templates for the Register Factory). If you are a template developer, have a look at the trigger configuration and at the corresponding available plug-in implementations of triggers, like e.g., Java Plug-in or XML Plug-in.

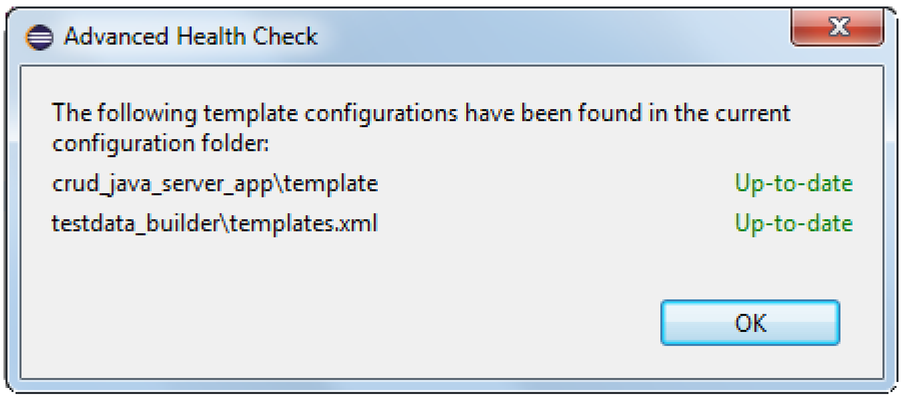
If all the checks are passed you see the following message:


Health Check

1. CobiGen configuration project 'CobiGen_Templates'... OK.
2. CobiGen context configuration 'context.xml'... OK.
3. Check validity of current selection... OK.

In this case everything is OK and the *Generate* button is not grayed out anymore so that you are able to trigger it and see the generation wizard.

In addition to the basic check of the context configuration, you also have the opportunity to perform an *Advanced Health Check*, which will check all available templates configurations (*templates.xml*) of path-depth=1 from the configuration project root according to their compatibility.


Advanced Health Check

The following template configurations have been found in the current configuration folder:

crud_java_server_app\template	Up-to-date
testdata_builder\templates.xml	Up-to-date

Analogous to the upgrade of the *context configuration*, the *Advanced Health Check* will also provide upgrade functionality for *templates configurations* if available.

8.3 Logging

If you have any problem with the CobiGen eclipse plug-in, you might want to enable logging to provide more information for further problem analysis. This can be done easily by adding the `logback.xml` to the root of the CobiGen_templates configuration folder. The file should contain at least the following contents, whereas you should specify an absolute path to the target log file (at the `TODO`). If you are using the [cobigen-templates](https://github.com/oasp/tools-cobigen/tree/master/cobigen-templates) project, you might have the contents already specified but partially commented.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- This file is for logback classic. The file contains the configuration for sl4j logging -->
<configuration>
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file><!-- TODO choose your log file location --></file>
        <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <Pattern>%n%date %d{HH:mm:ss.SSS} [%thread] %-5level %logger - %msg%n
            </Pattern>
        </encoder>
    </appender>
    <root level="DEBUG">
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

9. Template Development

9.1 Helpful links for template development

- FreeMarker Root Page
  - Expressions Cheat Sheet
  - Complete Language reference
  - FreeMarker Template Tester
- Variables to access Java source model