

OpenArchitectureWare 4.1 Recipe Framework Reference

Markus Voelter, voelter@acm.org, www.voelter.de

PRIMARY SPONSORS



Informatik AG



Enterprise
Software Solutions



SECONDARY SPONSOR

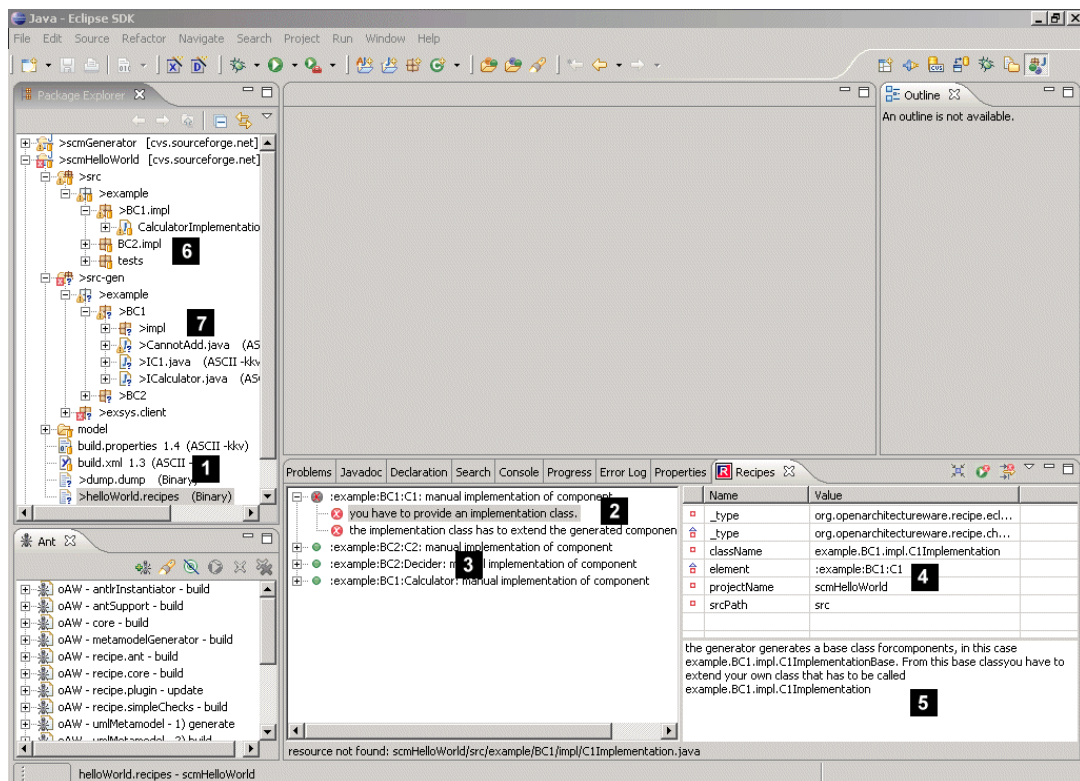
Table of Contents

INTRODUCTORY EXAMPLE AND PLUGIN.....	3
INSTALLING THE PLUGIN.....	5
USING CHECKS WITH YOUR OAW 3 GENERATOR.....	5
USING CHECKS WITH YOUR OAW 4 GENERATOR.....	7
USING ANT FOR CHECKING.....	8
IMPLEMENTING YOUR OWN CHECKS.....	9
HELLO WORLD.....	9
MORE SENSIBLE CHECKS.....	10
ECLIPSE CHECKS.....	11
MAKING CHECKS AVAILABLE TO THE ECLIPSE PLUGIN.....	12
FRAMEWORK COMPONENTS.....	13
LIST OF CURRENTLY AVAILABLE CHECKS.....	14

Introductory Example and Plugin

Currently it not feasible in MDS2 to generate 100% of an application. Usually, you generate some kind of implementation skeleton into which developers integrate their own manually written code. For example, the generator generates an abstract base class, from which developers extend their implementation classes – which contains the application logic.

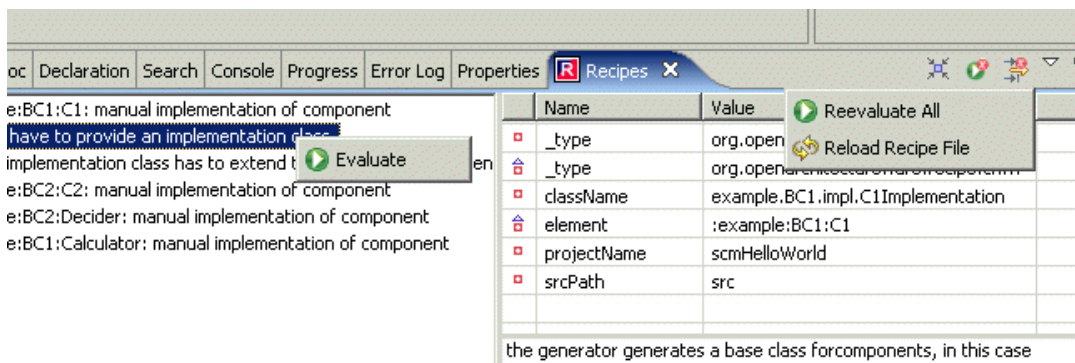
The screenshot above shows an example – and also illustrates the use of the Recipe Plugin. Let’s look at the various items in the screenshot.



- 1 The generator creates a so-called .recipes file. How this file is created will be shown below. The recipe file contains the checks that need to be executed on the code base. In the context menu of a .recipes file you’ll find an item called *Open Recipe File* that opens the checks in the respective browser. Note that the file has to have a .recipes extension – otherwise the plugin will not open the file!
- 2 Here you can see failed checks. The messages tell you that you have to write the

- implementation class, and (the second item) you'll have to make sure it extends from the generated base class.
- 3 Here you can see a check that worked well.
 - 4 Each check contains a number of parameters; the table shows the parameters of the selected check. For example the name of the class to be generated, the one from which you have to inherit, etc. This is basically for your information. If you double-click on a certain row, the value of the parameter is copied to the clipboard – and you can paste it to wherever you need.
 - 5 Here you can see a descriptive text that explains what you have to do in order to fix the failed check.
 - 6 Here is the manually written code (in the src folder). You can see the class CalculatorImplementation – this is why the check that verifies its presence is successful. There is no C1Implementation why the check that checks its presence fails.
 - 7 This is the generated code folder (src-gen). It contains the generated base classes.

There are a couple of options to work with those recipes, as shown in the next illustration.



If you right-click on a check in the treeview, you can reevaluate the check explicitly. The buttons at the top right of the view has three buttons: the first on collapses the tree view. The “play button with the small red cross” re-evaluates all the failed checks – and only those! The third button is a filter button; if selected, the tree view hides the checks that are ok. In the view’s drop down menu (activated with the small downward-pointing triangle) there are two entries: the green run-button labelled “reevaluate all” reevaluates all checks, even those that have been successful before. And finally, the reload button reloads the recipe file, and reevaluates everything.

There are two important automation steps:

- First of all, you should make sure that when you run the generator – typically through an ant file – the workspace will be refreshed after the ant run (you can configure this in Eclipse). If you do this, the view will automatically reload the recipe file and reevaluate all the checks.
- There are certain kinds of checks that automatically re-evaluate if the workspace changes. This means that, for example, if you add the implementation class in the above example, as soon as you save the class file, the check will evaluate to true. The question which checks will be evaluated automatically has to be defined in the check definition – see below.

Installing the Plugin

There are two steps. The first one installs the plugin itself, i.e. the Recipe Browser View, etc. The respective plugin is `org.openarchitectureware.recipe`. As usual you install it by copying it into the Eclipse plugin directory or just downloading it from the oAW update site. If the plugin is installed in this way, it can only evaluate the (relatively trivial) checks in the `recipe.simpleChecks` project. To check useful things, you'll have to extend the plugin – you have to contribute the checks that should be executed. For this purpose, the `org.openarchitectureware.recipe` plugin provides the *check* extension point. A number of useful checks that can be evaluated in Eclipse are contained in the `org.openarchitectureware.recipe.eclipseChecks` plugin. You should install that one, too. It also comes automatically from the update site.

In general, this means: whenever you develop your own checks and want to evaluate them in Eclipse, you *have* to contain them in a plugin and extend the above-mentioned extension point. Otherwise it will not work.

Referencing the JAR files

In order for the workflow to find the recipe JARs, they need to be on the classpath. The easiest way to achieve that is to make your generator project a plugin project and reference the recipe plugins in the plugin dependencies (all the oAW plugins with *recipe* in their name). This will update your classpath and add the necessary JARs. If, for some reason, you don't want your projects to be plugins, you have to reference the JAR files of the above mentioned plugins manually.

Using Checks with your oAW 3 generator

The easiest way to create a recipe file containing your checks is to write a `ModelModifier` that instantiates the tests and writes the checks file. Such a `ModelModifier` follows the following template:

```
public class ChecksMM implements ModelModifier {

    private String outdir;

    public ChecksMM( String outdir ) {
        this.outdir = outdir;
    }

    public int executionStrategy() {
        return RUN_AFTER_CHECK_CONSTRAINTS;
    }

    public void modifyModel() {
        // create checks here
        CheckRegistry.setChecksFileName( outdir+"/scm.recipes");
        CheckRegistry.dumpToFile();
    }
}
```

Since this approach is always roughly the same, we could have created this template as an abstract class for you to reuse. However, that would mean we'd have to reference the `ModelModifier` interface - and that would bind us to version 3.x or 4.x of oAW - and we wanted to make sure the framework runs with 3.x and 4.x. You can easily write the necessary class yourself.

Let's see how the example shown above is implemented. For the example, we first have to find all component instances since we have to issue a check for each one - there needs to be implementation code for each component.

```
public void modifyModel() {
    for (Iterator iter = SCMComponentExtent.findAll().iterator();
        iter.hasNext();) {
        SCMComponent c = (SCMComponent) iter.next();
```

We then add a so-called `ElementCompositeCheck`. `CompositeChecks` are checks that in turn contain other checks, according to the GoF Composite pattern. The `ElementCompositeCheck` has the additional property of containing a reference to a model element as well as an informative text. So as the first parameter, we pass the Component `c`, and then a text.

```
        ElementCompositeCheck ecc =
            new ElementCompositeCheck(c,
```

```
"manual implementation of component");
```

The text says, that you have to do something about the manual implementation of the component. The CompositeCheck doesn't do any checks itself, so we have to add the respective child checks. We'll do this now. We use an Eclipse-based `JavaClassExistenceCheck`. This is an Eclipse-specific check, since it uses Eclipse Java AST APIs as well as the workspace change notification mechanism. You have to pass four parameter:

- A message that shown up in the tree view in the plugin
- You then have to specify the Eclipse project, in which you expect the Java class
- Then you provide the name of source folder in which you expect the code
- Finally, you provide the fully qualified class name of the class you expect

```
JavaClassExistenceCheck javaClassExistenceCheck =
    new JavaClassExistenceCheck(
        "you have to provide an implementation class.",
        "scmHelloWorld", "src",
        c.FullyQualifiedManualImplementationClassName()
    );
```

You can also provide a longer text that shows up in the explanation box...

```
javaClassExistenceCheck.setLongDescription(
    "the generator generates a base class for"+
    "components, in this case "+
    c.FullyQualifiedImplBaseName()+
    ". From this base class you have to extend "+
    "your own class that has to be called "+
    c.FullyQualifiedManualImplementationClassName()
);
```

We then add this check to the composite check defined above.

```
ecc.addChild( javaClassExistenceCheck );
```

We then add a second check that verifies that the manually written class extends the generated base class. We pass the same four parameters as above, plus the fully qualified name of the base class. We then add this check to the composite check.

```
JavaSupertypeCheck javaSupertypeCheck =
    new JavaSupertypeCheck(
        "the implementation class has to extend the "+
        "generated component base class",
        "scmHelloWorld", "src",
        c.FullyQualifiedManualImplementationClassName(),
        c.FullyQualifiedImplBaseName()
    );
ecc.addChild( javaSupertypeCheck );
```

Since the composite check is a top level check (it's not part of another composite) we add it to the check registry... And finally, we set the file name and write it to disk.

```
        CheckRegistry.addCheck( ecc );
    }
    CheckRegistry.setChecksFileName( outdir+"/scm.recipes");
    CheckRegistry.dumpToFile();
}
```

That's all.

Using Checks with your oAW 4 generator

In openArchitectureWare Version 4 you have to write a workflow component that creates the recipes. Your custom workflow component has to extend the *RecipeCreationComponent* base class and overwrite the *createRecipes* operation. The operation has to return the collection of recipes that should be stored into the recipe file. In the workflow file, you then have to configure this component with the name of the application project in Eclipse, the source path where the manually written code can be found and the name of the recipe file to be written.

Please take a look at the *emfHelloWorld* example. It contains an extensive example of how to use the recipes in oAW 4.

Using Ant for Checking

You can also check the recipes using ant. Of course you cannot use those nice and cool interactive checks - and you also can't use Eclipse-based checks. They *can* be in the recipes file, since the ant task skips them automatically. The following piece of ant code shows how to run the checks - should be self-explanatory. Note that you have to use all the jar files from the recipe.ant project.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<project name="scm - hello world - generate" default="generate">
  <property file="build.properties" />

  <path id="ant.runtime.classpath">
    <pathelement location="${GENROOT}" />
    <fileset dir="${GENROOT}" includes="*.jar"/>
    <fileset dir="${OAWROOT}/dist" includes="*.jar"/>
    <fileset
      dir="${RECIPE.CORE.DIR}/dist"
      includes="*.jar"/>
    <fileset
      dir="${RECIPE.SIMPLECHECKS.DIR}/dist"
      includes="*.jar"/>
  </path>
</project>
```

```

<fileset dir="${RECIPE.ANT.DIR}/dist" includes="*.jar"/>
<fileset dir="${RECIPE.ANT.DIR}/lib" includes="*.jar"/>
<fileset dir="${RECIPE.ECLIPSECHECKS.DIR}"
    includes="*.jar"/>
</path>

<target name="check" depends="">
<taskdef name="check"
    classname="org.openarchitectureware.\
        recipe.ant.RecipeCheckTask">
    <classpath refid="ant.runtime.classpath" />
</taskdef>
<check
    recipeFile="L:/workspace/xy/helloWorld.recipes"/>
</target>

</project>

```

The checks use log4j logging to output the messages. So you can set the log level using the log4j.properties file. The following output shows all the checks being successful:

```

Buildfile: 1:\exampleWorkspace-v4\scmHelloWorld\build.xml
check:
    [check] 0    INFO - checking recipes from file:
                L:/runtime-EclipseApplication/xy/helloWorld.recipes
BUILD SUCCESSFUL
Total time: 1 second

```

If you set the log level to DEBUG, there's more; you can see that all the Eclipse checks are skipped.

```

Buildfile: 1:\exampleWorkspace-v4\scmHelloWorld\build.xml
check:
    [check] 0    INFO - checking recipes from file: L:/runtime-
                EclipseApplication/xy/helloWorld.recipes
    [check] 60   DEBUG - [skipped] resource exists exists -
                skipped - mode was batch only.
    [check] 70   DEBUG - [skipped] resource exists exists -
                skipped - mode was batch only.
    [check] 70   DEBUG - [skipped] resource exists exists -
                skipped - mode was batch only.
    [check] 80   DEBUG - [skipped] resource exists exists -
                skipped - mode was batch only.
    [check] 80   DEBUG - [skipped] resource exists exists --
                skipped - mode was batch only.
    [check] 90   DEBUG - [skipped] resource exists exists -
                skipped - mode was batch only.
    [check] 90   DEBUG - [skipped] resource exists exists -
                skipped - mode was batch only.
    [check] 90   DEBUG - [skipped] resource exists exists -

```

```
skipped - mode was batch only.  
BUILD SUCCESSFUL  
Total time: 1 second
```

If there are errors, they will be output as a log4j ERROR level message.

Implementing your own Checks

Hello World

The following piece of code is the simplest check you could possibly write:

```
package org.openarchitectureware.recipe.checks.test;  
  
import org.openarchitectureware.recipe.core.AtomicCheck;  
import org.openarchitectureware.recipe.eval.EvaluationContext;  
  
public class HelloWorldCheck extends AtomicCheck {  
  
    private static final long serialVersionUID = 1L;  
  
    public HelloWorldCheck() {  
        super( "hello world", "this check always succeeds" );  
    }  
  
    public void evaluate(EvaluationContext c) {  
        ok();  
    }  
  
}
```

A couple of notes:

- You can define any kind of constructor you want – passing any parameters you like. You *have* to pass at least two parameters to the super constructor: the first one is the name, a short name, of the check. The second parameter is the somewhat longer message. You can call `setLongDescription()` if you want to set the explanatory text.
- You can pass a third parameter to the super constructor: and that is one of the two constants in `EvalTrigger`. By default, the `EvalTrigger.ON_REQUEST` is used which means that the check is only evaluated upon explicit request. If you pass `EvalTrigger.ON_CHANGE`, the check will be automatically re-evaluated if the Eclipse workspace changes.
- You should define a serial version uid since Java serialization is used for the recipe file.
- In the `evaluate()` method you do the check itself. We will explain more on that later.

More sensible checks

More sensible checks distinguish themselves in two respects:

- First, you'll typically pass some parameters to the constructor which you'll store in member variables and then use in the evaluate() operation.
- You can store parameters for display in the table view in the plugin. You can use the setParameter(name, value) operation for that. More on that below.
- The evaluation will contain a certain business logic.

An example:

```
public void evaluate(EvaluationContext c) {
    if ( something is not right ) {
        fail( "something has gone wrong" );
    }
    if ( some condition is met ) {
        ok();
    }
}
```

By the way, you don't need to care about the EvaluationContext. Its only needed by the framework.

Eclipse Checks

Eclipse checks are a bit special. If the check were implemented in the way described above, you'd have a lot of dependencies to all the Eclipse plugins/jars. You'd have these dependencies as soon as you'd instantiate the check - i.e. also in the generator when you configure the check. In order to avoid this, we have to decouple the configuration of a check in the generator and its evaluation later in Eclipse:

1. During configuration we don't want any Eclipse dependencies since we don't want to "import" half of Eclipse into our ant-based code generator
2. However, when evaluating the check we obviously need the Eclipse dependencies, otherwise we couldn't take advantage of Eclipse-based checks in the first place.

An Eclipse check is thus implemented in the following way. First of all, our check has to extend the EclipseCheck base class.

```
public class ResourceExistenceCheck extends EclipseCheck {
```

Again, we add a serial version uid to make sure deserialization will work.

```
private static final long serialVersionUID = 2L;
```

In the constructor we decide whether we want to have this check evaluated whenever the Eclipse workspace changes (EvalTrigger.ON_CHANGE) or not. We also store some of the

parameters in the parameter facility of the framework. Note that we *do not* implement the *evaluate()* operation!

```
public ResourceExistenceCheck( String message,
    String projectName, String resourceName ) {
    super( "resource exists exists",
        message, EvalTrigger.ON_CHANGE );
    setProjectName( projectName );
    setResourceName( resourceName );
}

private void setProjectName(String projectName) {
    setParameter( "projectName", projectName );
}

private void setResourceName(String resourceName) {
    setParameter( "resourceName", resourceName );
}
}
```

In order to provide the evaluation functionality, you have to implement an *ICheckEvaluator*. *It has to have the same qualified name as the Check itself, postfixed with Evaluator*. During the evaluation of the check, the class is loaded dynamically based on its name. A wrong name will result in a runtime error during evaluation the Eclipse Plugin.

```
public class ResourceExistenceCheckEvaluator
    implements ICheckEvaluator {

    public void evaluate( AtomicCheck check ) {
    String projectName =
        check.getParameter("projectName").getValue().toString();
    String resourceName =
        check.getParameter("resourceName").getValue().toString();
    IWorkspace workspace = ResourcesPlugin.getWorkspace();
    IResource project =
        workspace.getRoot().getProject(projectName);
    if ( project == null )
        check.fail("project not found: "+projectName);
    IFile f = workspace.getRoot().getFile(
        new Path(projectName+"/"+resourceName) );
    String n = f.getLocation().toOSString();
    if ( !f.exists() ) check.fail(
        "resource not found: "+projectName+"/"+resourceName);
    check.ok();
    }
}
```

When implementing the evaluator, you can basically do the same things as in the `evaluate()` operation in normal checks. However, in order to set the `ok()` or `fail("why")` flags, you have to call the respective operations on the *check* passed as the parameter to the operation.

Making checks available to the Eclipse plugin

In order to allow the Eclipse runtime to execute your checks, it has to find the respective classes when deserializing the recipe file. This is a bit tricky in Eclipse, since each plugin has its own classloader. So assume you want to define your own checks and want to use them in Eclipse; what you have to do is: implement your own plugin that extends a given extension point in the Recipe Browser plugin. The following XML is the plugin descriptor of the `org.openarchitectureware.recipe.eclipseChecks.plugin.EclipseChecksPlugin`, a sample plugin that comes with the recipe framework and provides a number of Eclipse-capable checks.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
  id="org.openarchitectureware.recipe.eclipseChecks"
  name="%plugin_name"
  version="4.0.0"
  provider-name="%provider_name"
  class="org.openarchitectureware.recipe.\
    eclipseChecks.plugin.EclipseChecksPlugin">
```

Here we now define the jar file that contains our checks (will be important below!)

```
<runtime>
  <library name="oaw-recipe-eclipsechecks.jar">
    <export name="*" />
  </library>
</runtime>
```

The required plugins mainly depend on the implementations of the CheckEvaluators, of course, however, you have to make sure the dependencies contains the `org.openarchitectureware.recipe` plugin, since you're going to extend an extension point defined therein.

```
<requires>
  <import plugin="org.eclipse.ui" />
  <import plugin="org.eclipse.core.runtime" />
  <import plugin="org.eclipse.core.resources" />
  <import plugin="org.openarchitectureware.recipe" />
  <import plugin="org.eclipse.jdt.core" />
</requires>
```

This is the important line: here you specify that you extend the *check* extension point of the Recipe Browser Plugin. If you don't do this, deserialization of the recipe file will fail and you'll get nasty errors. And yes, you need the dummy element; otherwise the class loading "magic" will not work.

```
<extension point="org.openarchitectureware.recipe.check">
    <dummy/>
</extension>

</plugin>
```

When you need to use the checks outside of Eclipse (e.g. in the generator for configuration/serialization purposes) you just add the plugin's jar to your generator classpath. You *don't* need all the Eclipse plugin's jars referenced in the requires section, since these things will only be used during evaluation!

Framework components

Component	Plugin	Depends on	Description
recipe.core	Yes	-	Framework core. Needed whenever you do anything with recipes
recipe.ant	Yes	recipe.core	Contains the ant task to check recipes. Needed only for recipe evaluation in ant
recipe.simpleChecks	Yes	recipe.core	Contains a number of (more or less useful) sample checks
recipe.plugin	Yes	recipe.core	Contains the Eclipse Recipe Browser view
recipe.eclipsechecks.plugin	Yes	recipe.core	Contains the pre-packaged Eclipse checks.

List of currently available Checks

This table contains a list of all currently available checks. We are working on additional ones. Contributions are always welcome! This list might thus not always be up to date - just take a look at the code to find out more.

Type	Classname	Purpose
Batch	<i>org.openarchitectureware.recipe.checks. file.FileExistenceCheck</i>	Checks whether a given file exists
Batch	<i>org.openarchitectureware.recipe.checks. file.FileContentsCheck</i>	Checks whether a given substring can be found in a certain file
Eclipse	<i>org.openarchitectureware.recipe. eclipseChecks.checks. JavaClassExistenceCheck</i>	Checks whether a given Java class exists
Eclipse	<i>org.openarchitectureware.recipe. eclipseChecks.checks. JavaSupertypeCheck</i>	Checks whether a given Java class extends a certain superclass
Eclipse	<i>org.openarchitectureware.recipe. eclipseChecks.checks. ResourceExistenceCheck</i>	Checks whether a given Eclipse Workspace Resource exists

About our Sponsors

itemis GmbH & Co. KG is an independent IT service company with an emphasis on consulting, coaching, and software development. Every single itemis expert provides many years of project experience and widespread knowledge about all object oriented and component based software development issues - especially in the field of model driven software development.

b+m is the founder of the openArchitectureWare project. The software was originally developed within the scope of many successful projects. b+m opened the software to the community in late 2003. All of the paradigms of Model-Driven Software Development including Product Line Engineering and not only the generator framework have become a key concept for product and customer specific development at b+m. b+m customers can make use of long time experience and substantial know-how in that field. Located at the company headquarters in Melsdorf/Kiel and at its subsidiaries in Berlin, Cottbus, Hamburg, Hanover and Kiel the b+m staff of 205 provides practical solutions for customized business applications, business process optimization and comprehensive architecture, project and quality management.

oose Innovative Informatik GmbH offers coaching, consulting and training in all themes about software engineering. The main focus of their activities are software architecture, requirements engineering and project-management. oose have first-hand information and experience, because our staff take actively part with others in actual trends, standards and innovations. Our staff support this and pass their know-how regularly on by writing and publishing books or being speaker at conferences, etc. Within the OMG oose collaborate actively on the specifications of the UML and also the SysML.

MID Enterprise Software Solutions GmbH is a leading supplier of optimized tool environments for standardsbased and model-centric software development as well as business process modeling. This includes professional tool consulting and tool components to build a complete tool environment using the best techniques and tool modules available - Architectural and Operational Excellence. With innovatorAOX, MID provides a holistic standard tool environment for object- and function-oriented software development as well as business process and data modeling to help its customers establish highly efficient processes and tool environments for software production, The unique and seamless integration of business process modeling into the development process ensures an unprecedented level of convergence of business requirements and implemented IT systems. Project members from all departments speak the same language and all requirements are clearly described.