Institute for Software Technology

UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

The common Ant script
(Version 1.0 – June 25, 2011)

Sascha Strauß

strauss@uni-koblenz.de

Contents

1 Introduction

Apache Ant is used to build complex Java projects. It is quite useful for automating tasks such as creating the project's jar file or performing the JUnit tests. Ant uses declarative scripts that are written in XML. These scripts declare so-called *target*s. The targets uses *tasks* for performing the desired operations.

This document describes the so-called *common Ant script* that is used for projects depending on JGraLab. The common Ant script serves as general build file for such projects. It replaces the old Ant script that was used before. The old Ant script did not exploit all of Ant's features for generalizing Ant scripts. When creating specific Ant scripts, it was required to declare all required targets and call the targets from the old Ant script. This produced large and uncomfortable specific Ant scripts that were hard to maintain. The goal of the new common Ant script is making specific Ant scripts shorter and easier to maintain.

This document targets users of JGraLab who want to create their own projects depending on JGraLab. Readers should be familiar with the Ant terminology.

The common Ant script provides all common features an Ant script for projects depending on JGraLab requires. A specific Ant script for such a project is derived from the common Ant script using Ant's import mechanism. In most cases, the specific Ant script only needs to set some properties. Listing 7 on page 19 in appendix A shows a minimal specific Ant script.

For some projects, it requires more adjustments. However, these adjustments are still less complex than in the old Ant script.

Section 2 describes the non-target-specific properties that can be set in specific Ant scripts. Section 3 describes how the classpath is set in the common Ant script. Section 4 describes the targets that are defined in the common Ant script and their properties. Section 5 describes advanced techniques for customizing a specific Ant script.

2 Properties

This section describes general rules for properties and introduces the general properties that are not target specific. Target-specific properties are introduced in section 4.

2.1 Overview of general properties

The following list gives an overview of the general properties.

projectname contains the name of the project. By convention, this must be identical to the project directory and may only contain lowercase letters.
The default value is empty.

basePackage contains the name of the base package.
The default value is "de.uni_koblenz.${projectname}".

basePackagePath contains the path to the base package.
The default value is "de/uni_koblenz/${projectname}".

main contains the name of the main class (the class whose main method is executed, if the jar file is called directly).
The default value is empty, so it has to be overridden.

main.fq contains the fully qualified name of the main class.
The default value is "${basePackage}.${main}".

maxmemsize contains the maximum amount of memory for tasks that use a forked VM.
The default value is "512M".

minmemsize contains the minimum amount of memory for tasks that use a forked VM.
The default value is "256M".

The following list shows all properties that contain directory information. By convention, the default values of these properties should not be overridden.

project.dir contains the relative path to the project directory.
The default value is "../${projectname}".

src.dir contains the relative path to the source directory.
The default value is "${project.dir}/src".

build.dir contains the relative path to the build directory.
The default value is "${project.dir}/build".

classes.dir contains the relative path to the compiled Java classes.
The default value is "${build.dir}/classes".

common.dir contains the relative path to the directory of the project common.
The default value is "../common".

There are further target specific properties with directory information that can be found in section 4.

2.2 The location of property definitions

Properties in Ant are comparable to constants. Once set, they are immutable. However, it is possible to override properties in specific Ant scripts. Properties for overriding default values have to be defined before the import clause for the common Ant script.

Properties that are used as variables in other property declarations have to be defined before they are used. This means custom properties, that use standard properties from the common Ant script, have to be defined after the import clause.

Please note, that these rules imply that defining properties overriding default values while using other properties with default values is only possible if the used default values are explicitly set before the import clause. Since this relation is transitive, the described scenario should be avoided.

3 Classpath

The common Ant script provides two classpaths. These classpaths require the following properties.

comlib.dir contains the relative path to the common libraries.
The default value is `"${common.dir}/lib"`.

lib.dir contains the relative path to the project specific libraries.
The default value is `"${project.dir}/lib"`.

jgralab.location contains the location of JGraLab's jar file.
The default value is `"../jgralab/build/jar/jgralab.jar"`.

ist_utilities.location contains the location of the jar file including the IST utilities.
The default value is `"${comlib.dir}/ist_utilities/ist_utilities.jar"`.

The first and more important classpath is called `classpath` and contains all important libraries that are required for compiling and using the project. It includes all compiled java classes, all project-specific libraries, the IST utilities and JGraLab. It also includes the path `classpathExtension` which has to be defined in specific Ant scripts. Unlike the overriding of property definitions, this path has to be defined after the import statement. If libraries from the common library directory are required, these have to be added to `classpathExtensions`. If no further entries are required, `classpathExtensions` may be left empty. However, it always has to be declared.

Listing 1 shows an example on how common libraries should be added to the classpath.

Listing 1: Adding common libraries to the classpath

```
1  <path id="classpathExtension">
2      <fileset dir="${comlib.dir}/apache/commons/cli/"
           includes="**/*.jar" />
3      <fileset dir="${comlib.dir}/jdom/" includes="**/*.jar" />
4  </path>
```

Please note that the jar files are not added directly to the classpath, but all jar files in the containing directory. If the name of a jar file changes (e.g. due to a new version number), this method avoids the necessity to update all specific Ant scripts of projects that depend on the changed jar file.

The second classpath is called `testclasspath` and contains `classpath`, the compiled test cases and JUnit's jar file for executing the test cases. This classpath cannot be directly extended.

4 Targets

This section describes the targets provided by the common Ant task.

4.1 Targets for compiling and generating

Here all targets that are used for compiling source code and for generating other artifacts are introduced. The most important target is `build`, which is also the default target for every specific Ant script. The target `build` first compiles other required projects.

4.1.1 Targets for building required projects

All projects that use the common Ant script automatically depend on JGraLab and on the IST utilities. The common Ant script provides targets that call the Ant scripts of these projects automatically. The target for building JGraLab is called `jgralab`. The target for building the IST utilities is called `ist_utilities`. The jar files of these projects are only built if they do not already exist.

After these targets have been executed, the build target executes the target `clean` for removing most generated files that might have been created by a previous run of the Ant script (see section 4.2 for details). Then the target `compile` is invoked (see section 4.1.3). This target compiles the actual source code of the project. Since most projects require a custom graph schema, targets for creating the Java files that represent this schema have to be generated.

4.1.2 Schema related targets

A schema can be specified by either exporting an RSA model to an XWI file or by providing a TG file. JGraLab can convert xmi files to tg files. In both cases, a tg file is used for generating Java source files representing the schema.

The target for generating Java source files from tg files is called `generateschema`. It is controlled by the following parameters.

schema.file contains the relative path to the tg file.

By default this property is unset. Only if it is set to a value, the target `generateschema` is actually executed.

schema.location contains the relative path to the source directory, where the generated Java files should be stored.
The default value is "${src.dir}".

schema.implementationMode contains a comma separated list of implementation modes that should be generated.
The default value is "standard". E.g. if additional transaction support is required, it should be overridden with the value "standard,transaction". Valid values may contain (so-far) standard, transaction, savemem and database.

schema.withoutTypes corresponds to the flag -w in TgSchema2Java. If it is set to "true", type specific methods will not be created in the schema files.
The default value is "false".

schema.subtypeFlag corresponds to the flag -f in TgSchema2Java. If it is set to "true", additional type specific methods will be created, that have a flag for deciding if subtypes should be considered (default behavior) or not. It is ignored, if **schema.withoutTypes** is set to "true".
The default value is "false".

The target for converting an exported XMI file to TG is called convertschema. If an XMI file is specified, this target is executed first. It is controlled by the following parameters.

xmi.schema.file contains the relative path to the XMI file.
By default this property is unset. Only if it is set to a value, the target convertschema is actually executed. This also requires **schema.file** to be set, because otherwise the target generateschema would not be executed and the build would most likely fail.

rsa2tg.f corresponds to the flag f in Rsa2Tg. If it is set to "true", the name of from roles is used for creating undefined edge class names.
The default value is "false".

rsa2tg.u corresponds to the flag u in Rsa2Tg. if it is set to "true", all unused domains will be deleted before generating the TG file.
The default value is "false".

rsa2tg.n corresponds to the flag n in Rsa2Tg. If it is set to "true", the navigability info will be interpreted as reading direction.
The default value is "false".

After the schema files have been generated, the target compile can actually compile the source files.

4.1.3 The target `compile`

The target compile is controlled by the following parameters.

compileincludes contains a list of Ant patterns that specify which source files are being included in the compile process.
The default value is empty, which means everything is included.

compileexcludes contains a list of Ant patterns that specify which source files are being excluded from the compile process.
The default value is empty, which means, nothing is excluded.

javac.targetVM contains the compatibility level of the class files.
The default value is `"1.6"`.

debug Declares if debug information should be included in the class files.
The default value is `"false"`.

debuglevel contains a comma separated list of which debug information will be included, if **debug** is set to `"true"`.
The default value is `"lines"`. Valid values may contain `lines`, `vars` and `source`.

After the compilation of the source code is done, the build target executes the targets for creating the jar file.

4.1.4 Targets for creating the jar file

The common Ant script is designed for creating standalone jar files. To achieve this, all required libraries are extracted and put into the project's jar file. The target `unjar` extracts all required libraries to a temporary folder.

The target `unjar` by default extracts the file `ist_utilities.jar` and all project specific libraries. It is controlled by the following properties.

tmp.dir contains the relative path to a non-existing directory that will be created, will serve as temporary directory and will be deleted during the build process.
The default value is `"${build.dir}/tmp"`.

unjar.disabled controls whether the target `unjar` is executed or not.
By default this property is unset. If it is set to an arbitrary value, the target `unjar` is not executed.

unjarexcludes contains a comma separated list of Ant patterns that specify which project specific libraries will not be contained in the project's jar file.
The default value is empty, which means, nothing is excluded.

The target `unjar` is often overridden by specific Ant scripts. Details on that can be found in section 5.1.1.

For actually building the output jar file, the target `jar` is used. It creates a jar file containing all files from the class folder, all non-source files from the source folder (e.g. image files) and all files that were extracted from the target `unjar`. It also deletes the temporary folder specified by the property **tmp.dir**. It can be controlled by the following properties.

jar.dir contains the relative path to the location of the project's jar file.
The default value is `"${build.dir}/jar"`.

resource.excludes contains a comma separated list of Ant patterns that specify which resources from the source directory will not be contained in the project's jar file.
The default value is empty, which means, nothing is excluded.

All other targets that are implicitly executed by the target `build` can be found in section 4.3.

4.2 Targets for cleaning

This section introduces the targets that are used for removing generated artifacts. There are two main targets for this purpose, `clean` and `cleanall`. The target `clean` removes all compiled class files, all compiled test cases, all generated schema files and the temporary folder, in case it was not deleted by a failed build attempt. The target `cleanall` removes everything that is removed by the target `clean` and additionally removes the jar file, the javadoc, the test results and calls the target `cleanall` from the Ant script that builds the IST utilities. The latter is done for convenience, so the IST utilities are never required to be rebuilt manually.

If the building of a project fails, the first step for resolving the problem is updating the projects `common`, `jgralab`, the project itself and all other projects that are required by it. Then the target `cleanall` should be invoked for at least the project itself and `jgralab`. Afterwards the Ant script for the project can be run normally.

The target `clean` calls the target `deleteGeneratedSchemaFiles`, which is responsible for deleting the Java files that have been generated by the target `generateschema`. It is only invoked if the property **schema.file** is set.

The target `cleanall` depends on the the target `deleteConvertedSchemaFile`, which deletes the tg file, specified by the property **schema.file**, if the property **xmi.schema.file** is set.

4.3 Additional targets

This section introduces all remaining targets.

4.3.1 The target `createClassesDir`

The target `createClassesDir` only creates the directory for the class files specified by the property **classes.dir**. It is implicitly called by `build`.

4.3.2 The target `customAntTasks`

The target `customAntTasks` depends on the target `jgralab`. It ensures that JGraLab's jar file is built and defines the Ant tasks `tgschema2java`, `deletegeneratedschema` and `rsa2tg`. These Ant tasks are implemented in JGraLab. All targets that use these tasks have to depend on `customAntTasks` rather than depend directly on `jgralab`. This can be important when overriding targets. It is implicitly called by `build`.

4.3.3 The target `ensureJarExists`

The target `ensureJarExists` only invokes the target `build`, if the project's jar file does not exist yet. This is a very useful target for avoiding unnecessary build cycles. However, this target cannot detect updates in the source code.

4.3.4 The target `sourcejar`

The target `sourcejar` creates a second jar file, based on the normal one. It makes a copy of the normal jar file and adds all Java source files from the source folder to it. It depends on the target `ensureJarExists`.

4.3.5 The target `document`

The target `document` creates the javadoc for the project. It can be controlled by the following properties.

doc.dir contains the relative path to the location of the project's javadoc files.
The default value is `"${build.dir}/doc"`.

documentexcludes contains a list of Ant patterns that specify which classes are not being added to the javadoc.
The default value is empty, which means, nothing is excluded.

document.access contains the information down to which visibility level the javadoc is generated.
The default value is `"public"`. Valid values are `"public"`, `"protected"`, `"package"` and `"private"`.

4.3.6 The target `run`

The target `run` invokes the main method of the main class. The main class is defined by the property **main.fq**. The target can be controlled by the following properties.

run.args contains the arguments that are passed to the main method.
The default value is empty.
run.jvmargs contains the arguments that are passed to the jvm.
The default value is empty.
run.dir contains a reference to the directory the application is executed from.
The default value is set to `"${project.dir}"`.

4.3.7 The target `test`

The target `test` compiles the test cases and runs JUnit tests. It can be controlled by the following properties.

testcases.dir contains the relative path to the source files of the junit test cases.
The default value is `"${project.dir}/testit"`.
testclasses.dir contains the relative path to the compiled junit test cases.
The default value is `"${build.dir}/testclasses"`.
testresults.dir contains the relative path to the directory that should contain the test results.
The default value is `"${build.dir}/testresults"`.
test.suite contains the fully qualified name of the test suite to be executed.
By default this property is unset. Only if it is set to a value, the target `test` is actually executed.
test.formattertype contains a string representing the style of the test results.
The default value is set to `"brief"`. Valid values are `"brief"`, `"plain"`, `"xml"` and `"failure"`.

4.3.8 The target `modify`

The target `modify` changes the build id of the project in the specified main class. It can be controlled by the following properties.

main.src contains a reference to the path of the source file of the main class.
The default value is set to `"${src.dir}/${basePackagePath}/${main}.java"`.

4.3.9 The target `addLicenseHeaders`

The target `addLicenseHeaders` adds a license header to every Java source file in the source folder. Existing headers will be replaced by the new one. It depends on the target `clean` for avoiding setting a header to the generated schema source files. It can be controlled by the following properties.

license.file contains a reference to the path of the file containing the license header that will be added to all Java files.
By default this property is unset. Only if it is set to a value, the target `addLicenseHeaders` is actually executed.

5 Further customizations of specific Ant scripts

Section 2 described how the custom Ant script can be customized by overriding properties. Section 3 showed how the classpath can be extended. Both of these features are sufficient for many cases. However, if a project is more complex or has more complex dependencies, it is most likely required to further customize its specific Ant script. This section shows where such customization might be required and describes the conventions that should be obeyed.

5.1 Customizing targets

Any specific Ant script, that imports the common Ant script, does not only include its source code.This import operation also includes an inheritance relation between the common Ant script and a specific Ant script. This relation particularly concerns the targets defined in the common Ant script.

It also incorporates a mechanism that is comparable to polymorphy in object oriented languages. The targets inherited from the common Ant script can be overridden. Also the targets defined by the common Ant script can be called. The details about these features and how they can be used for customizing a specific Ant script are described in the following.

It can be feasible to add targets to a specific Ant script. These targets can be used for creating callable targets that handle additional jobs which are not concerned with the build process.

5.1.1 Overriding inherited targets

A target can be overridden by just creating a target with the same name in the specific Ant script. This has two effects. The obvious effect is, if the target is called directly or indirectly by dependency in the specific Ant script, the overridden variant is invoked. The less obvious effect is, if a target in the common Ant script depends on a target that has been overridden in a specific Ant script, also the overridden variant is invoked. This behavior is comparable to polymorphic method calls in Java and allows a very flexible way of manipulating the behavior of a specific Ant script.

Listing 2 shows the `build` target of the common Ant script.

Listing 2: Target `build` in the common Ant script

```
1    <target name="build" depends="jgralab,clean,compile,jar" />
```

Assuming the project of a specific Ant script has defined a new target `pre_compile` that should be invoked before the target `compile` is invoked. For achieving this, the build target can be overridden as shown in listing 3.

Listing 3: Target `build` in a specific Ant script

```
1    <target name="build" depends="jgralab,clean,pre_compile,compile,jar" />
```

Here the build target is overridden and only the dependency has changed. This solution has one problem. If the target `compile` is called directly, the target `pre_compile` is not invoked. For this purpose, it would be feasible to let `compile` depend on `pre_compile`. For achieving this, the target `compile` has to be overridden. However, the original target `compile` should also be invoked, but without copying its content from the common Ant script. Ant provides a mechanism to call the targets from imported Ant scripts. The property `name` from the root tag `project` is used as identifier. This is comparable to the super reference in Java. In the common Ant script this attribute has the value `"common"`.

Listing 4 shows how the target `compile` can be overridden by using the original target.

Listing 4: Overriding the target `compile`

```
1    <target name="compile" depends="pre_compile,common.compile" />
```

Please note that the order of the required targets in the attribute `depends` dictates the order the targets are invoked. If the desired target can also be invoked after the inherited target, defining a new target can be avoided. E.g., this technique can be used for overriding the target `unjar` because the order the libraries are unpacked is not important.

The following example assumes the common library for command line parsing (Apache commons CLI) is required in the project.For creating a standalone jar file, only adding this library to the classpath is not sufficient. Its content has to be added to the project's jar as well. The common target `unjar` does not consider libraries from `classpathExtension`. So the target `unjar` should be extended for adding this behavior.

Listing 5 shows how this is done.

The target depends on the target `unjar` from the common Ant script (line 1). The unjar task (lines 2 - 4) unpacks all jar files that are found in the specified location into the temporary directory. The jar files are refered to in analogy to extending classpaths (see section 3).

Listing 5: Overriding the target `unjar`

```
1    <target name="unjar" depends="common.unjar">
2        <unjar dest="${tmp.dir}">
3            <fileset dir="${comlib.dir}/apache/commons/cli/"
                 includes="**/*.jar" />
4        </unjar>
5    </target>
```

5.1.2 Convention for target dependencies

This section shows some conventions for the dependencies of targets.

The common Ant script is designed to build a project, when the target `build` is called. The target `clean` should only clean the generated class files and schema files, but not the jar file, the test results and the documentation. This behavior should never be changed.

All targets that are invoked by `build` should also work when invoked directly. However, it might be better if they do not depend on each other. E.g. the target `jar` does not depend on `compile`. If a resource is changed (e.g. an image file), but the source code remains unchanged, it is better if the `jar` target only rebuilds the jar file using the class files that are already compiled instead of rebuilding the whole project.

Targets can be split into multiple targets if a single target would be too complex. A prominent candidate for this is the target `compile`. In case of `compile`, the target `common.compile` should be overridden completely, meaning the overriding target does not depend on the overridden target.

When overriding a target completely, it is important to preserve the behavior of the overridden target as well as possible. This means, the target has to depend at least on the same targets as the overridden target. E.g. when overriding `compile`, this means the target has to depend on the targets `createClassesDir` and `generateschema`.

5.2 Handling multiple projects and their dependencies

This section introduces concepts on how to create specific Ant scripts that handle the dependency relations with other projects. Primarily, the conventions described in this chapter concern only projects that use the common Ant script. However, it is also applicable to other projects with certain adjustments.

5.2.1 Project dependencies to other projects using the common Ant script

Some projects that use the common Ant script also depend on other projects using this script. In particular, this is true for projects that include a schema that is used by multiple projects.

A project depending on another project has to include the required project's jar file in the classpath. It also has to be able to create this jar file if it is not present.

Listing 6 shows the complete specific Ant script for the project `jgwnlclient` that depends on the project `jgwnl`. Please note that the property **unjar.disabled** is set to `"true"` for the required project `jgwnl`.

Listing 6: specific Ant script of the project `jgwnlclient`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <project name="jgwnlclient" basedir="." default="build">
4
5      <property name="projectname" value="jgwnlclient" />
6      <property name="main" value="JGWNLClient" />
7      <property name="basePackage"
           value="de.uni_koblenz.${projectname}.client" />
8
9      <import file="../common/common.xml" />
10
11     <property name="jgwnl.location" value="../jgwnl/build/jar/jgwnl.jar" />
12
13     <path id="classpathExtension">
14         <pathelement location="${jgwnl.location}" />
15     </path>
16
17     <target name="build" depends="jgwnl,clean,compile,jar" />
18
19     <target name="jgwnl">
20         <Ant dir="../jgwnl" antfile="build.xml" inheritAll="false"
               target="ensureJarExists" />
21     </target>
22
23     <target name="unjar" depends="common.unjar">
24         <unjar src="${jgwnl.location}" dest="${tmp.dir}" />
25     </target>
26
27 </project>
```

The path to the jar file of `jgwnl` is stored in a new property **jgwnl.location** (line 11). By convention, this property should always be called ***requiredProjectname*.location**. This location is added to the `classpathExtension` (line 14). The target `unjar` is overridden as described in section 5.1.1. The specific Ant script contains a new target `jgwnl` that builds the project `jgwnl` if its jar file does not exist yet (lines 19-21). This target calls the target `ensureJarExists` of the project `jgwnl`. By convention, this should be done for other projects in analogy. This ensures that the build is only invoked if the required project's jar file does not exist yet. The target `build` is overridden and includes this target in its dependency list (line 17).

If the project depends on multiple other projects, each of these projects requires a location property and a target that builds it if required. The target `build` has to be adjusted so that all these targets appear in the dependency list.

5.2.2 Project dependencies to other projects not using the common Ant script

For describing a dependency to a project that does not use the common Ant script, the convention is similar. However, third party projects should be put into the library folder indicated by **lib.dir**. If this is not feasible, the conventions introduced in section 5.2.1 have to be modified.

There has to be a property pointing to the project's jar file(s). This location has to be included in `classPathExtension` and should be considered by the target `unjar`. There also has to be a target building the required project. If this project provides a similar mechanism as the target `ensureJarExists`, this target should be called. If not, the project's main target has to be called (the target that corresponds to `build`). However, the disadvantage is, that the required project is always build, when the dependent project is build. If the required project's jar file can be modified, it is better to implement a target `ensureJarExists` for it.

A A minimal specific Ant script

Listing 7 contains a specific Ant script for a minimal HelloWorld application containing a schema. This file can be found in the directory `minimal` in the project `common`. It can serve as starting point for other specific Ant scripts.

Listing 7: Minimal specific Ant script

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <project name="de.uni_koblenz.minimal" basedir="." default="build">
4
5      <property name="projectname" value="minimal" />
6      <property name="main" value="HelloWorld" />
7      <property name="schema.file" value="MinimalSchema.tg" />
8
9      <import file="../common/common.xml" />
10
11     <path id="classpathExtension">
12
13     </path>
14
15 </project>
```

By convention, the attribute `name` has to be set to the value `packageName.projectName` (line 3). The properties **projectname** and **main** are overridden (lines 5-6). The property **schema.file** is set to the tg file of a minimal schema (line 7). If no schema is required, this line can be removed. The `classpathExtension` for this project is empty, because no further libraries are required.

When creating projects, please keep in mind to create a directory `lib`, because this is required in the classpath defined by the common Ant script.