Enterprise Computing: Ant: A Java build tool

Stephen Gilmore

The University of Edinburgh

eXtreme programming (XP) with Ant

Ant provides a framework that enables developers to rapidly configure processes for all phases of the software life cycle. An Ant build process is described in an XML file, called a buildfile.

eXtreme programming (XP) with Ant

Ant provides a framework that enables developers to rapidly configure processes for all phases of the software life cycle. An Ant build process is described in an XML file, called a buildfile.

The buildfile is used to configure the steps necessary to complete a build process. Users also can control the behavior of Ant throughout the build process by means of various options, properties, and environment variables.

This includes parameters such as the volume of information generated during a build, its form, and its destination.

This includes parameters such as the volume of information generated during a build, its form, and its destination.

It's even possible to cause certain actions to happen, such as sending an e-mail notification when a build completes.

This includes parameters such as the volume of information generated during a build, its form, and its destination.

It's even possible to cause certain actions to happen, such as sending an e-mail notification when a build completes.

Ant has a gentle learning curve, which makes it easy to create a useful tool without extensive knowledge of Ant.

Elements of a buildfile

Ant is directed to perform a series of operations through a buildfile. A buildfile is analogous to a makefile for make, and it is written in XML.

Elements of a buildfile

Ant is directed to perform a series of operations through a buildfile. A buildfile is analogous to a makefile for make, and it is written in XML.

Buildfiles are composed of projects, targets, and tasks. These constructs are used to describe the operations that a buildfile will perform when Ant is invoked.

Buildfile relationships

Projects, targets, and tasks have a hierarchical relationship.

Buildfile relationships

Projects, targets, and tasks have a hierarchical relationship.

- A buildfile describes a single project.

Buildfile relationships

Projects, targets, and tasks have a hierarchical relationship.

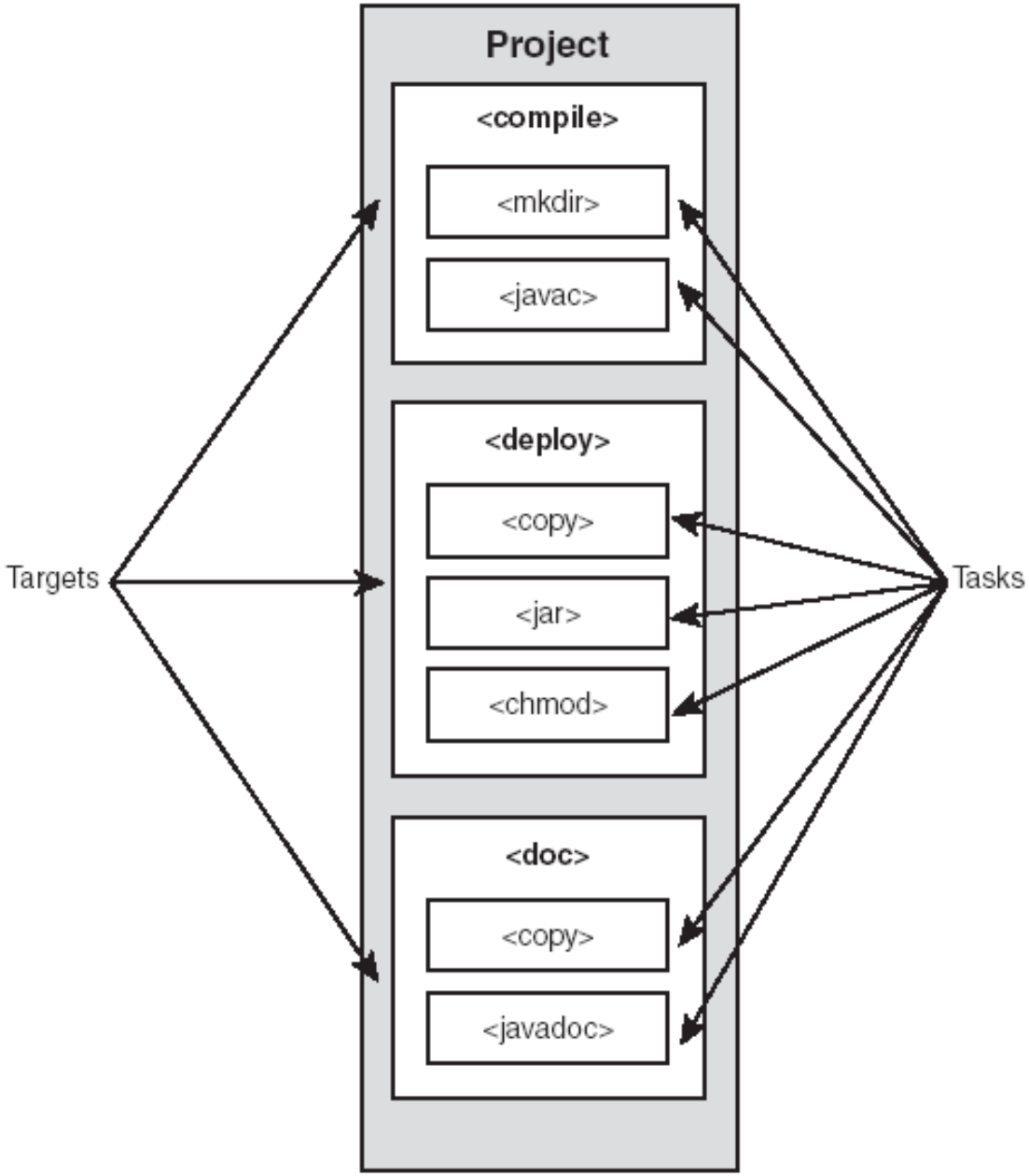
- A buildfile describes a single project.
- Within the single project are one or more targets.

Buildfile relationships

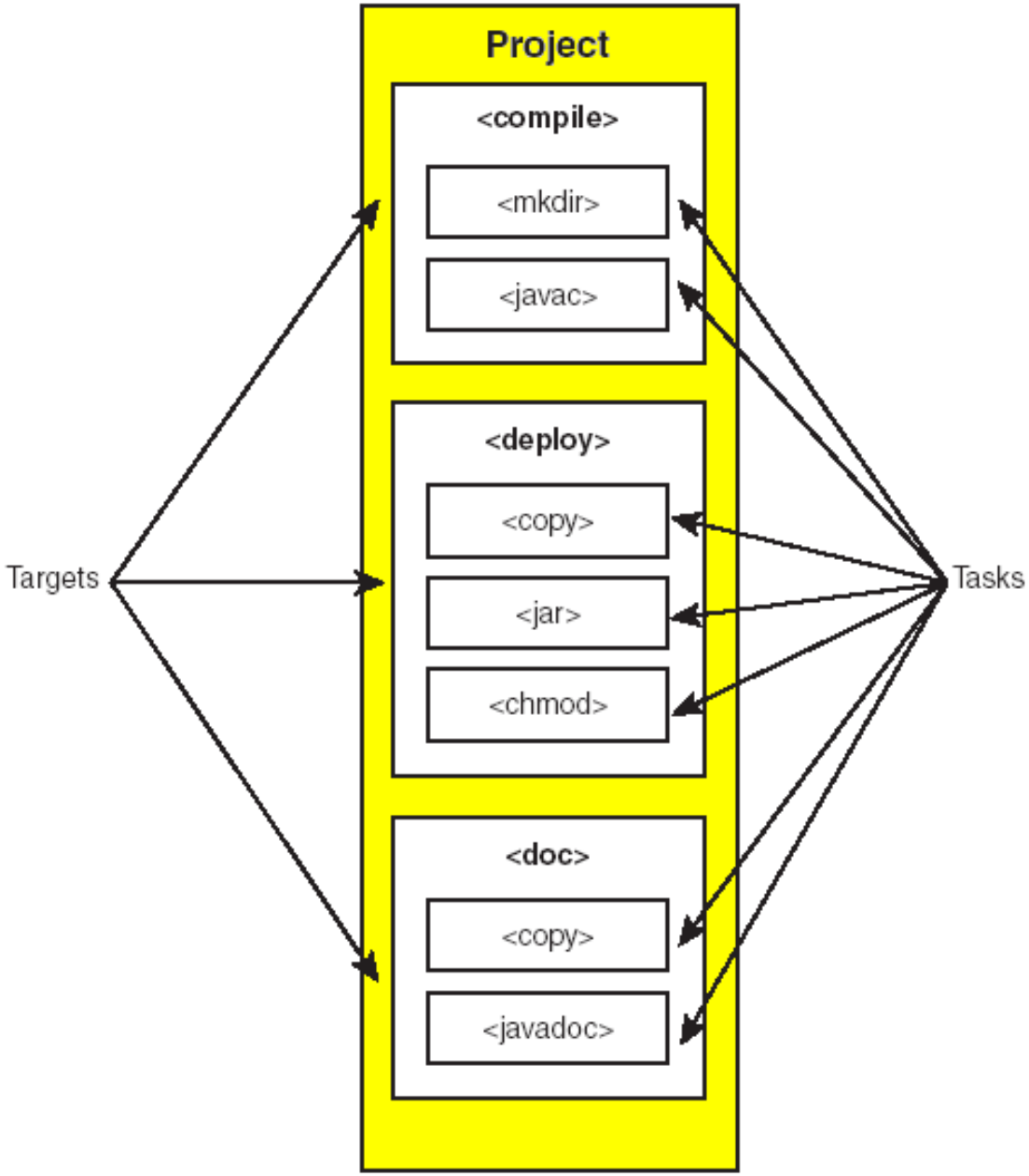
Projects, targets, and tasks have a hierarchical relationship.

- A buildfile describes a single project.
- Within the single project are one or more targets.
- Targets are composed of one or more tasks.

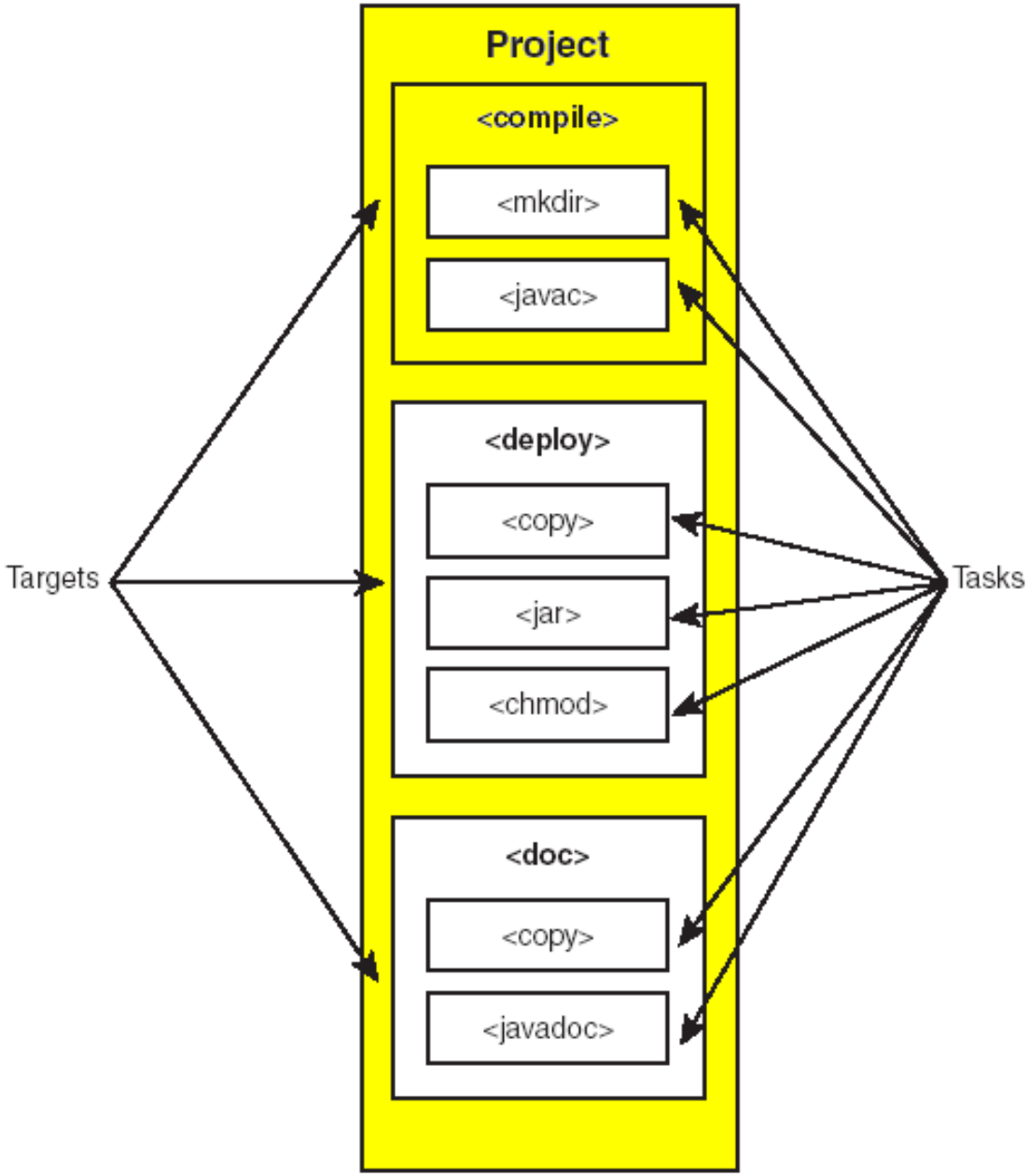
Projects, targets and tasks



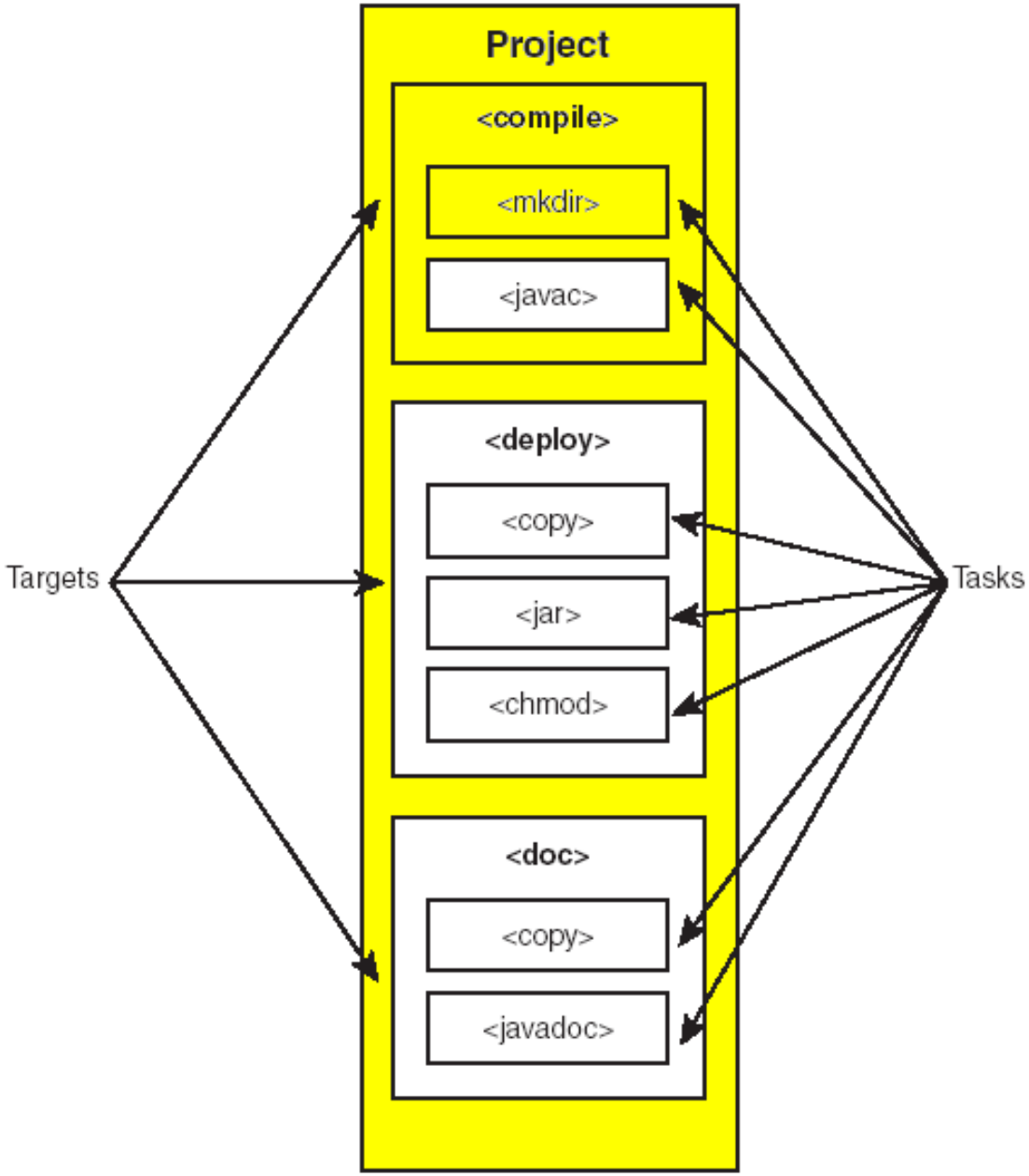
Projects, targets and tasks



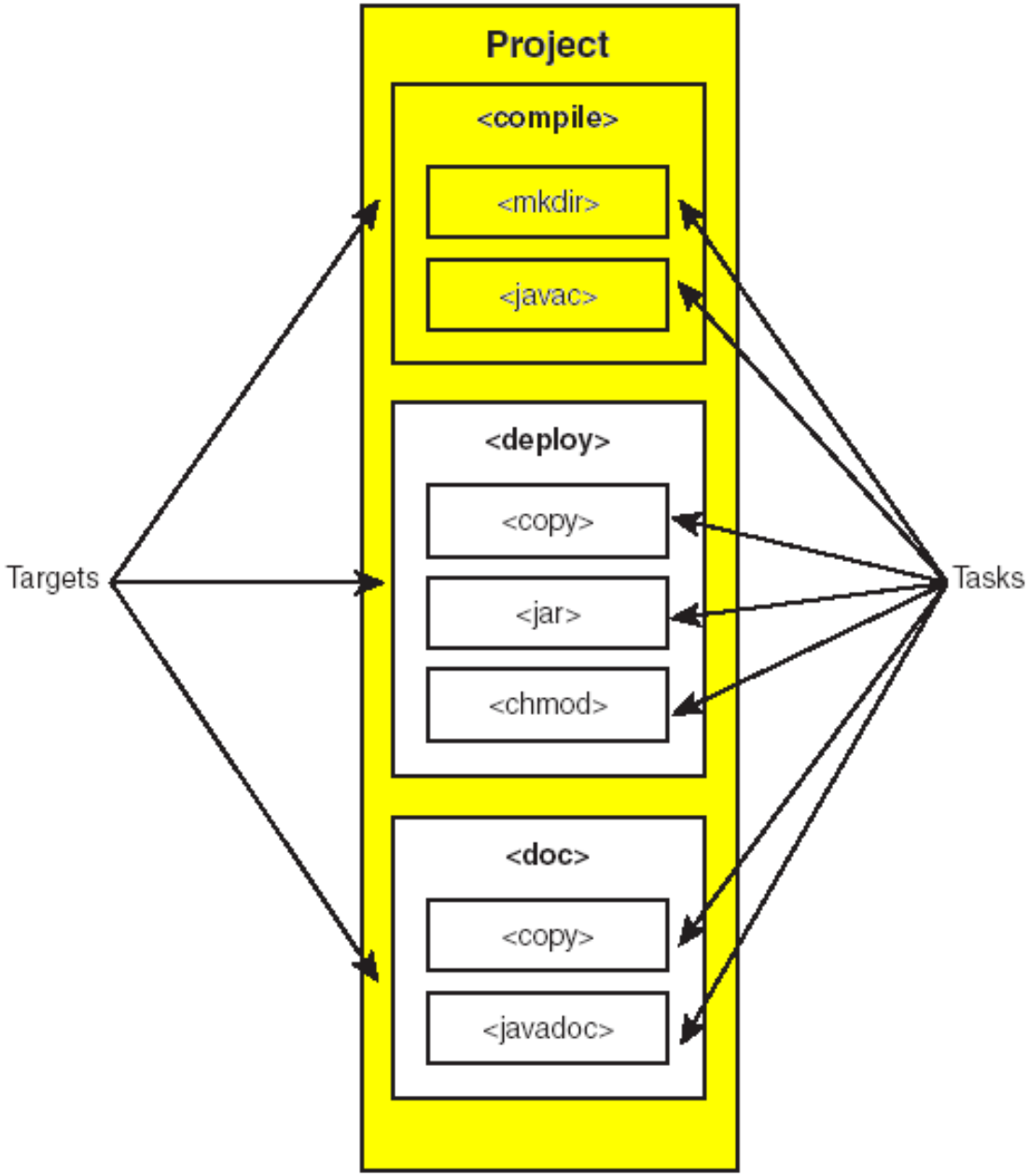
Projects, targets and tasks



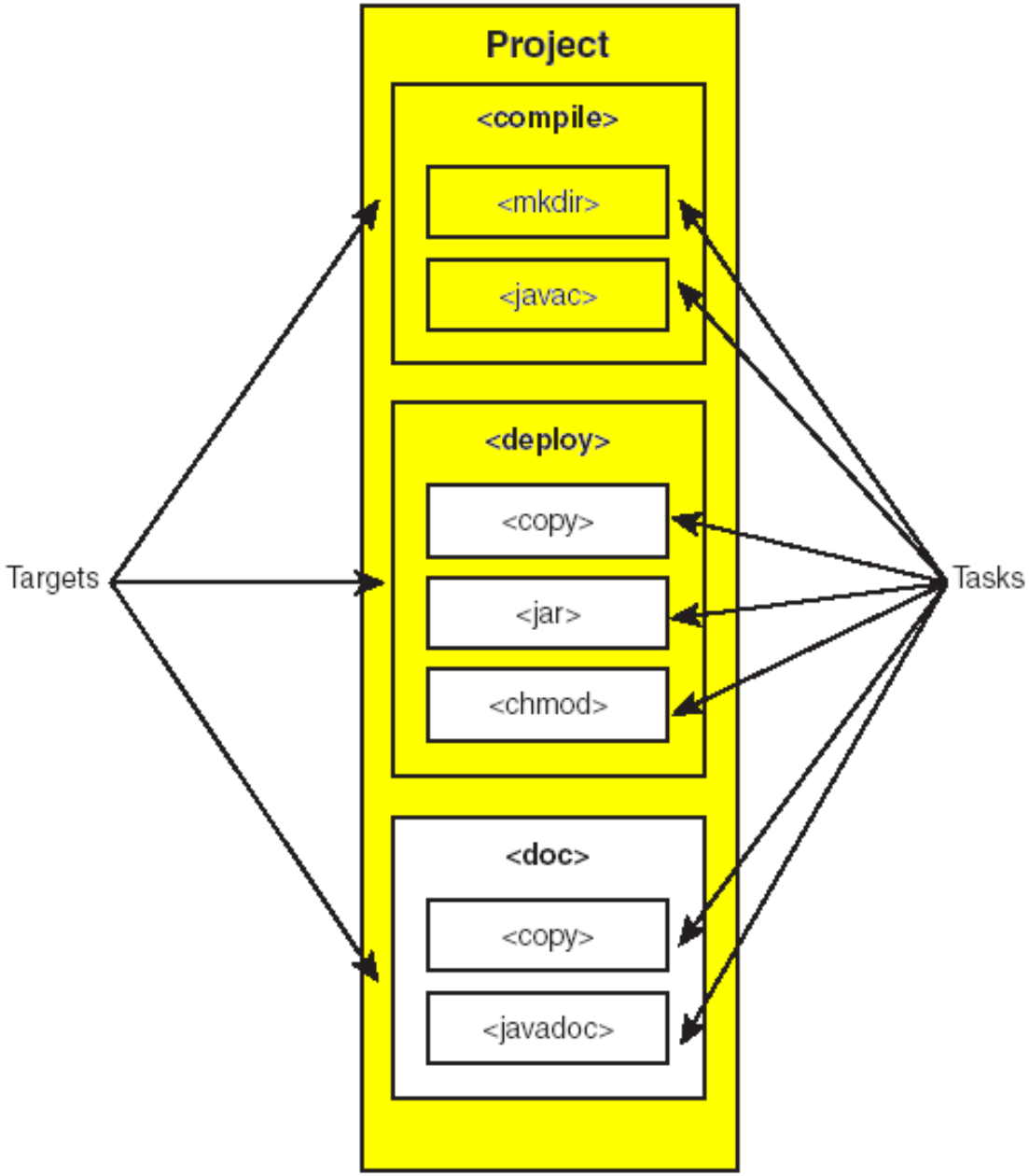
Projects, targets and tasks



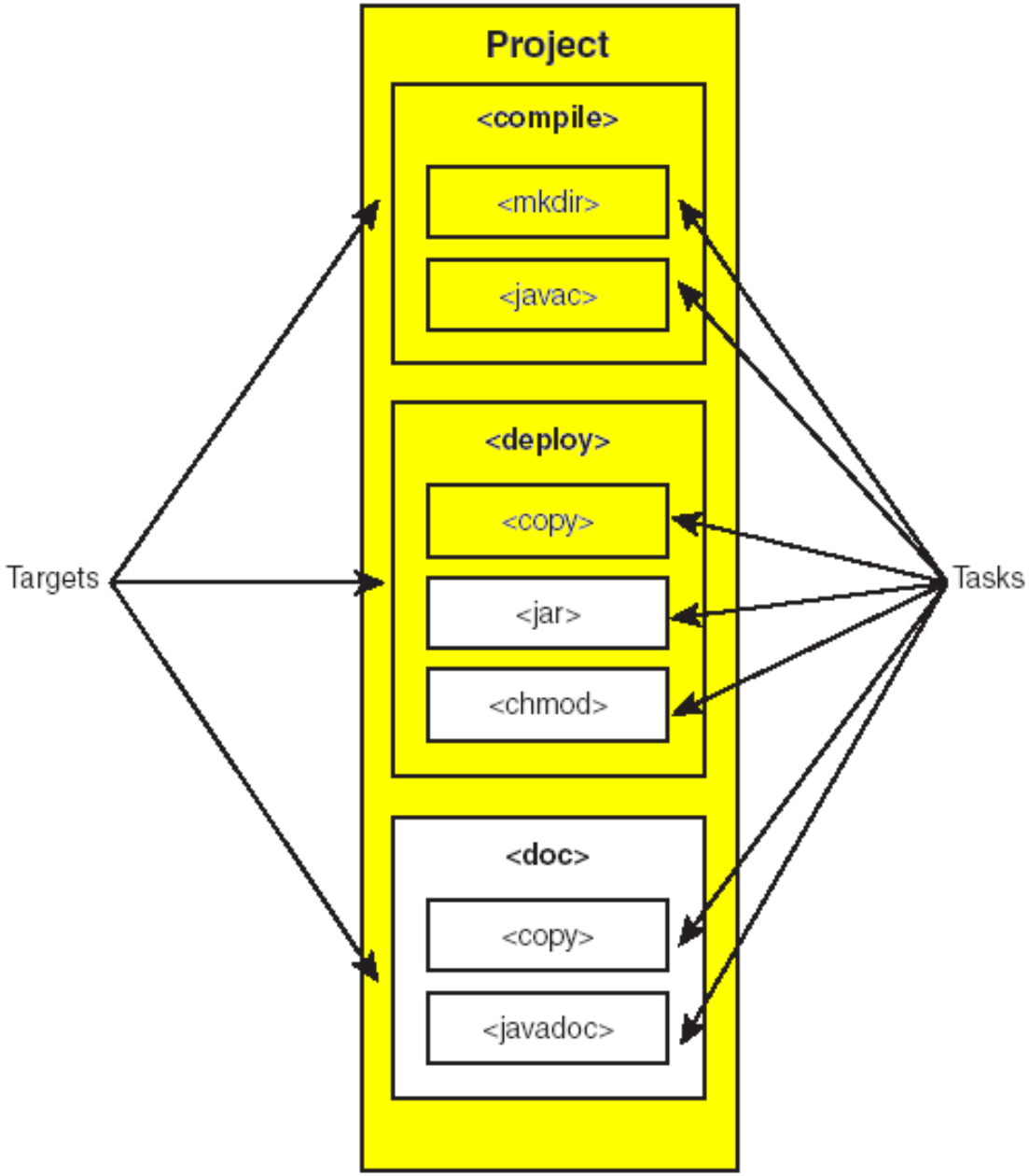
Projects, targets and tasks



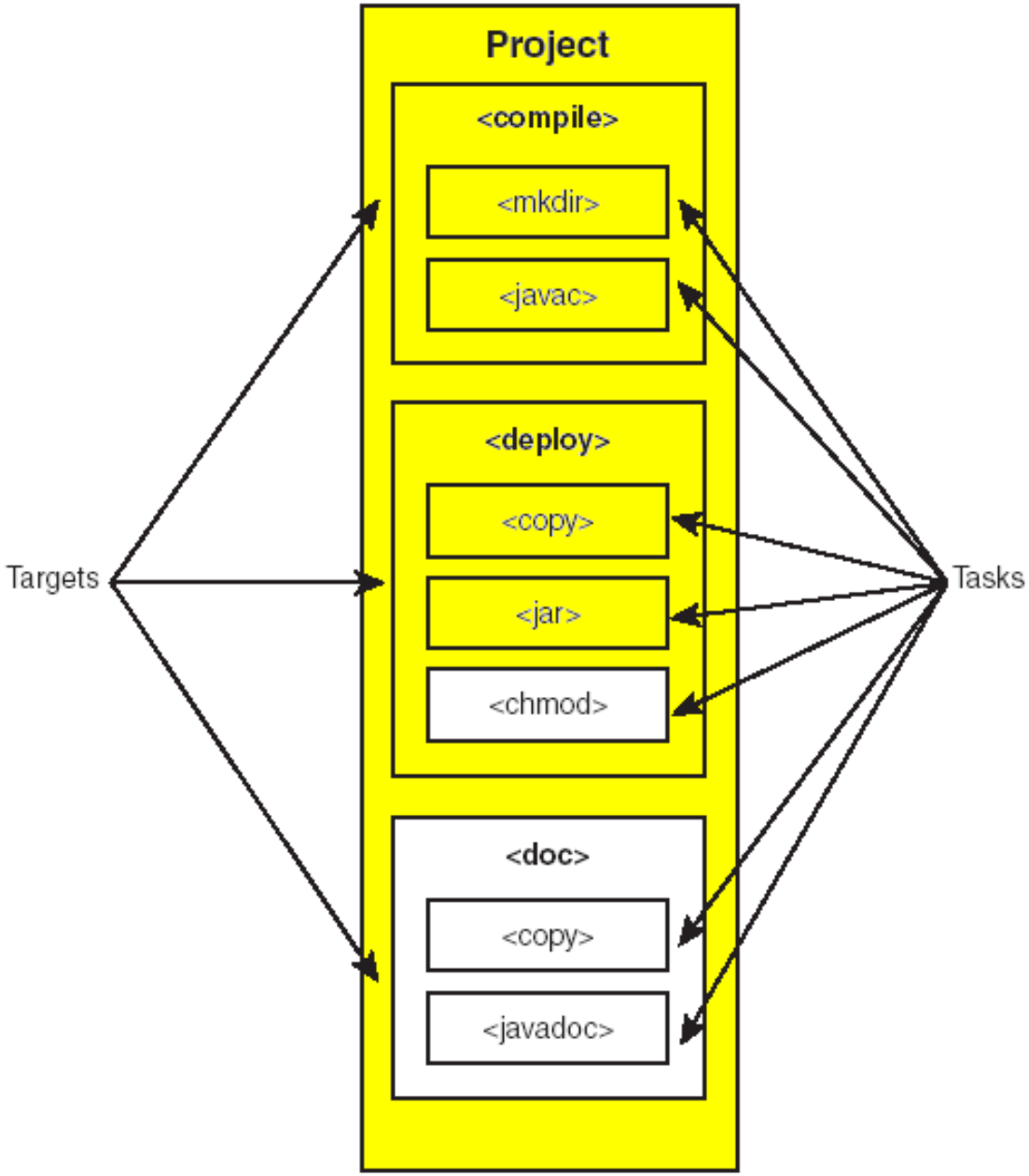
Projects, targets and tasks



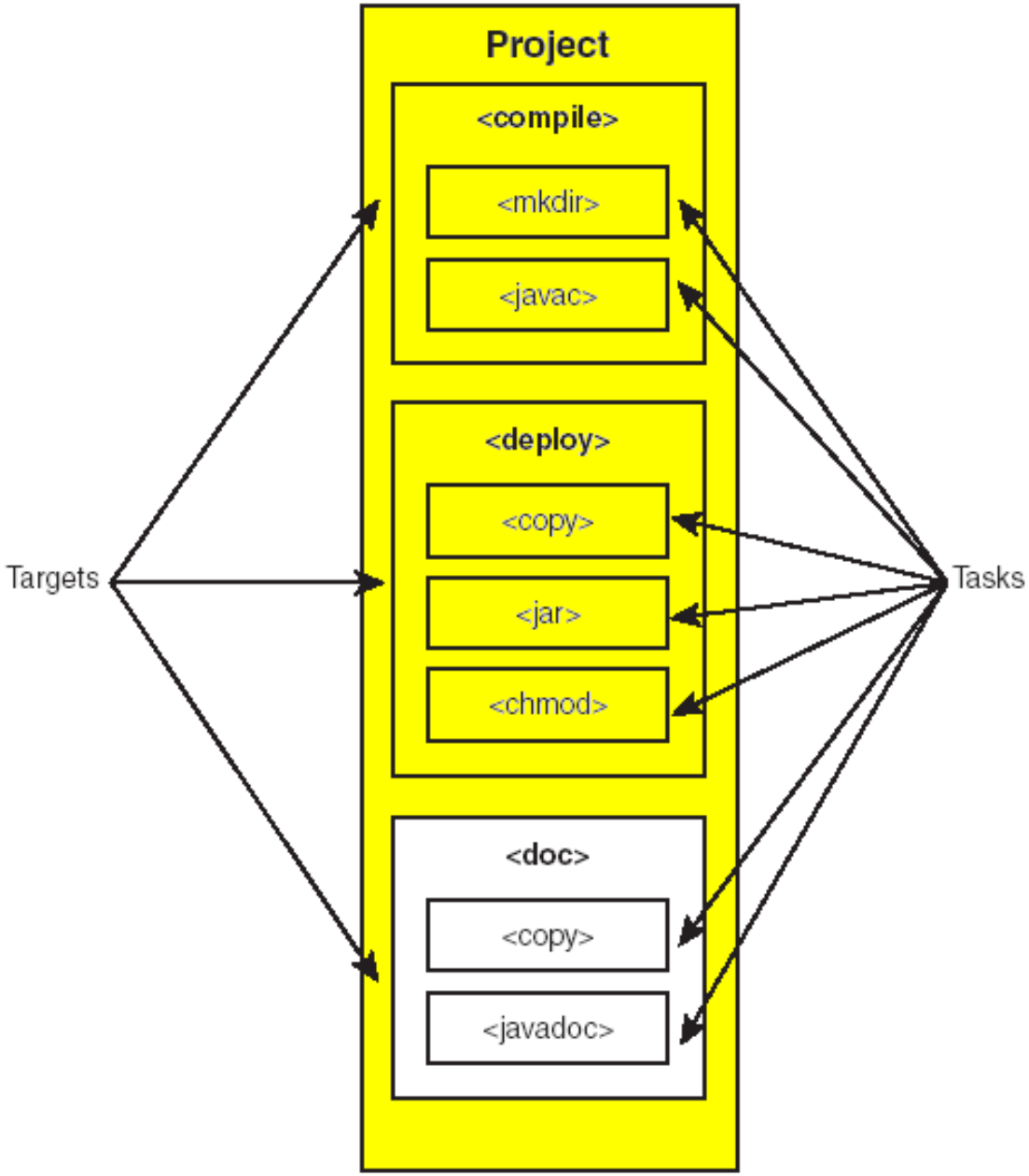
Projects, targets and tasks



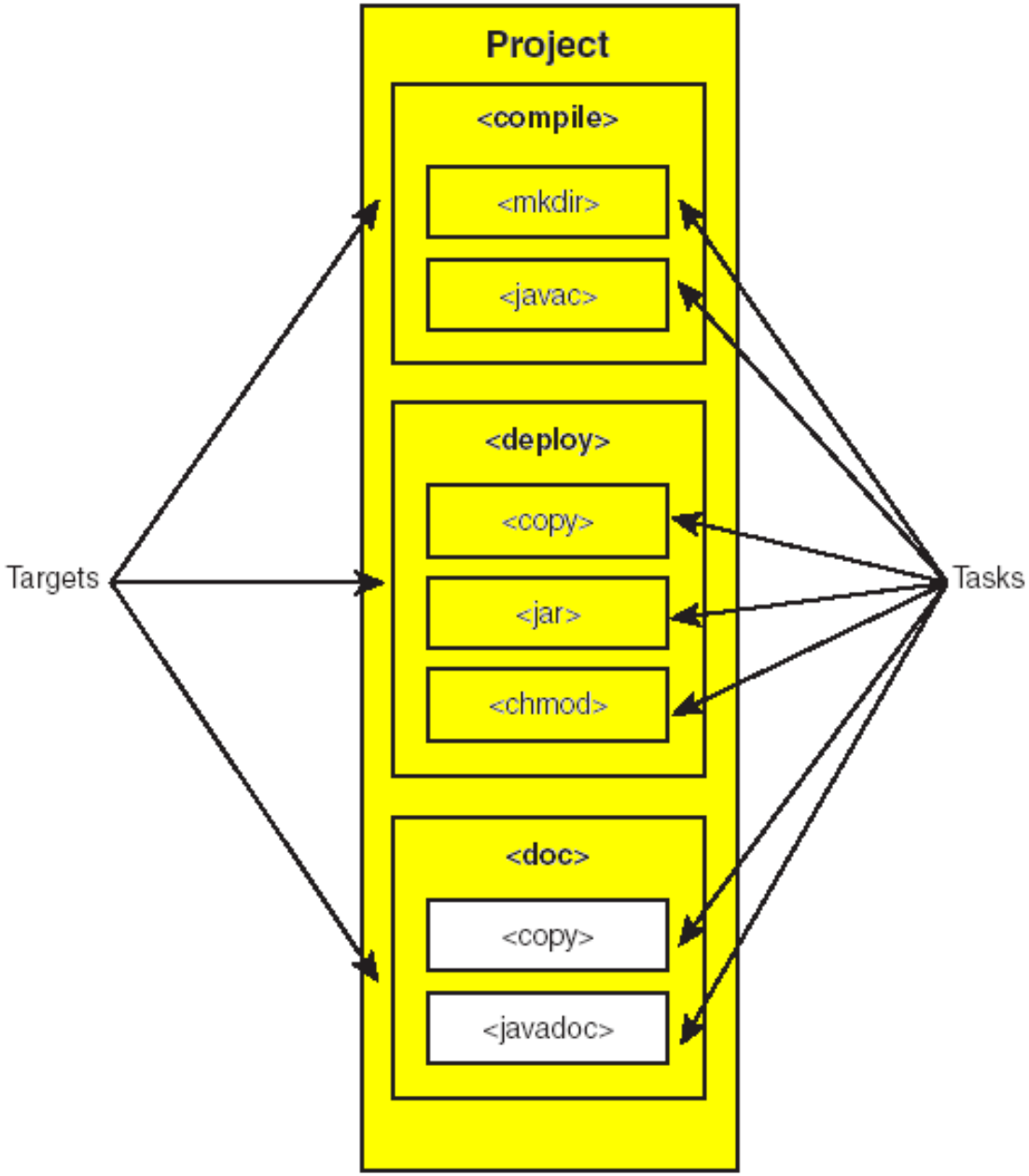
Projects, targets and tasks



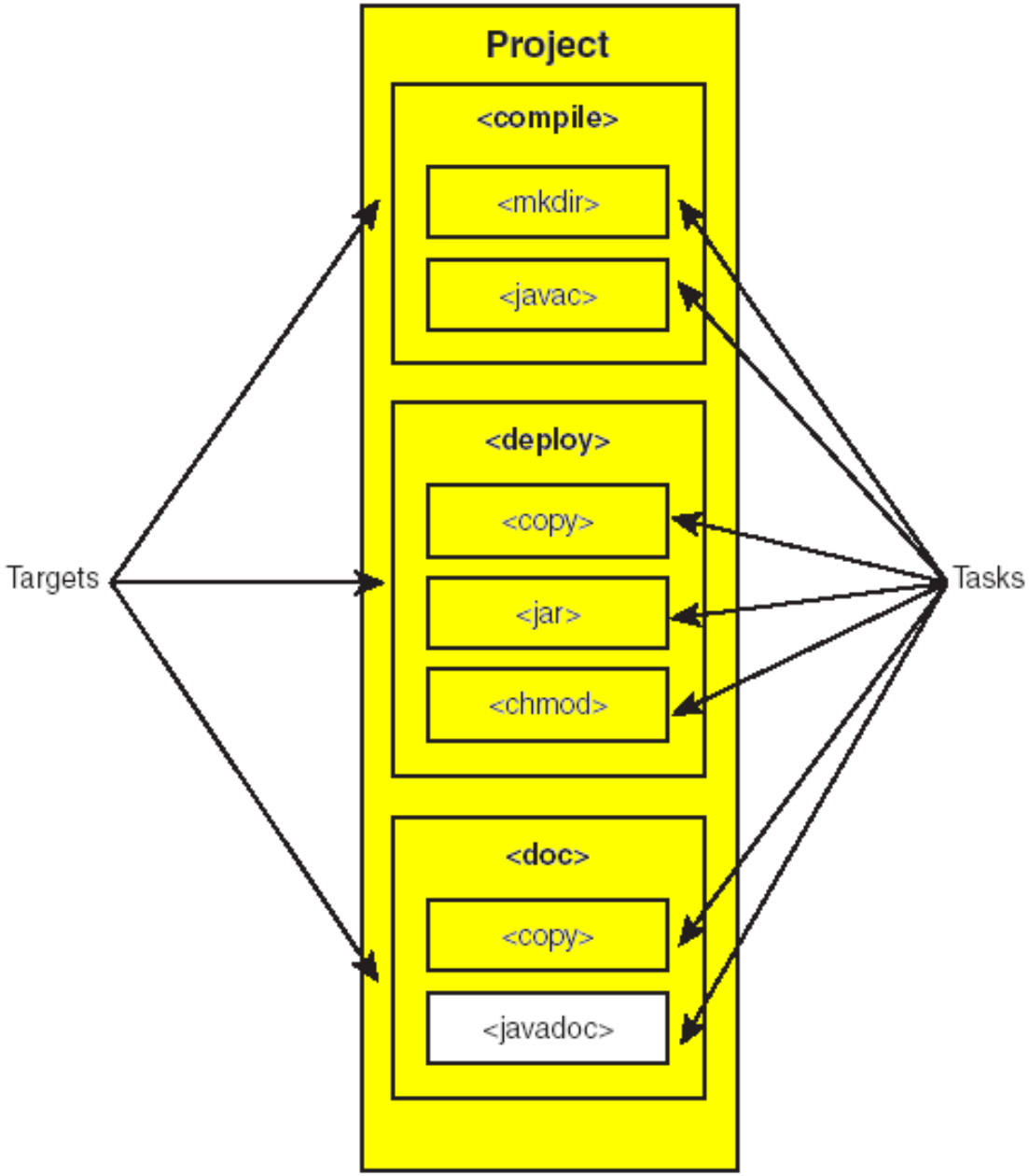
Projects, targets and tasks



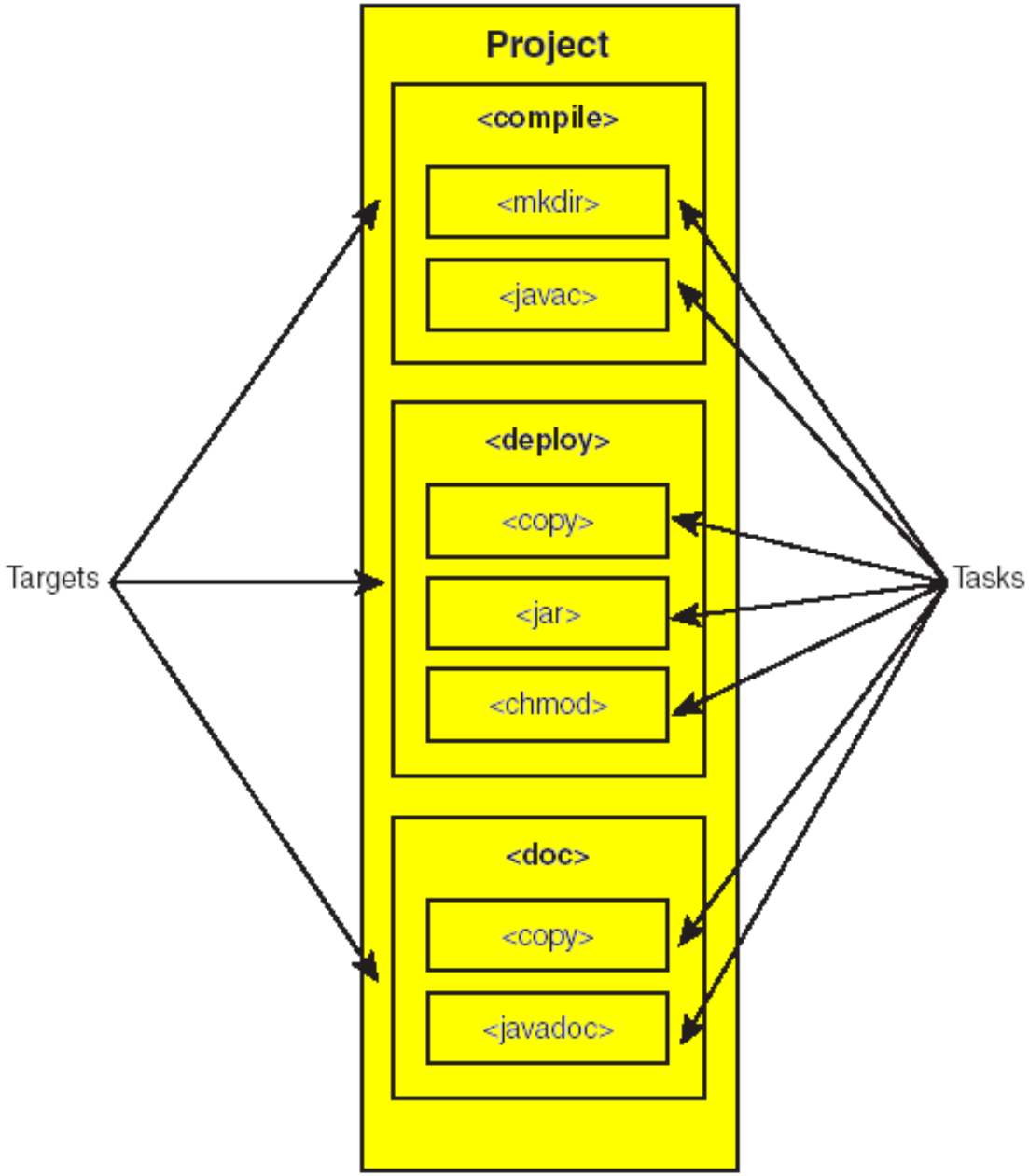
Projects, targets and tasks



Projects, targets and tasks



Projects, targets and tasks



Framework of the buildfile

The first step is to construct the basic framework of the buildfile, named build.xml. The buildfile can be called anything you like, but there's a reason for using the name build.xml. By default, Ant searches for a file by this name if no other buildfile name is passed to it at invocation.

Framework of the buildfile

The first step is to construct the basic framework of the buildfile, named build.xml. The buildfile can be called anything you like, but there's a reason for using the name build.xml. By default, Ant searches for a file by this name if no other buildfile name is passed to it at invocation.

If the file is named something else, simply pass the name on the command line with the -buildfile option.

A skeletal buildfile (build1.xml)

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <project name="myproject" default="test.ant"
3          basedir=".">
4 </project>
```

A skeletal buildfile (build1.xml)

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <project name="myproject" default="test.ant"
3          basedir=".">
4 </project>
```

```
[beetgreens]stg: ant -buildfile build1.xml
Buildfile: build1.xml

BUILD FAILED
Target 'test.ant' does not exist in this project.

Total time: 2 seconds
```

Targets

A target is a set of steps to be executed together as a unit. A target tag has five attributes: name; depends; if; unless; and description.

Targets

A target is a set of steps to be executed together as a unit. A target tag has five attributes: name; depends; if; unless; and description.

The name attribute is simply the name assigned to the target to refer to it, and this attribute is required.

Targets

A target is a set of steps to be executed together as a unit. A target tag has five attributes: name; depends; if; unless; and description.

The name attribute is simply the name assigned to the target to refer to it, and this attribute is required.

The depends attribute is used to define a dependency by one target on one or more other targets.

The description attribute enables you to create a descriptive comment.

The description attribute enables you to create a descriptive comment.

The attributes if and unless are used to define conditional execution on a target.

Effect of target failures

If execution of a target fails, the buildfile can be configured to either continue execution of the remaining targets or to stop the build process.

Effect of target failures

If execution of a target fails, the buildfile can be configured to either continue execution of the remaining targets or to stop the build process.

With a target, however, if any step fails, the rest of the target is abandoned.

Effect of target failures

If execution of a target fails, the buildfile can be configured to either continue execution of the remaining targets or to stop the build process.

With a target, however, if any step fails, the rest of the target is abandoned.

So, the execution of a target is not an atomic operation.

A skeletal buildfile (build2.xml)

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <project name="myproject" default="test.ant"
3         basedir=".">
4   <target name="test.ant"
5         description="A simple build file to test ant.">
6   </target>
7 </project>
```

```
[beetgreens]stg: ant -buildfile build2.xml
Buildfile: build2.xml

test.ant:

BUILD SUCCESSFUL
Total time: 1 second
```

```
[beetgreens]stg: ant -projecthelp -buildfile build
Buildfile: build2.xml
Main targets:

 test.ant  A simple build file to test ant.

Default target: test.ant
```

Tasks

The constructs that make up an Ant target are called tasks. Tasks are predefined operations that Ant can perform. The actual task implementation is a Java class.

Tasks

The constructs that make up an Ant target are called tasks. Tasks are predefined operations that Ant can perform. The actual task implementation is a Java class.

The behaviour of any given task is configured within the buildfile through attributes of the task.

Tasks

The constructs that make up an Ant target are called tasks. Tasks are predefined operations that Ant can perform. The actual task implementation is a Java class.

The behaviour of any given task is configured within the buildfile through attributes of the task.

Ant has two categories of tasks:

- core tasks; and
- optional tasks.

Core tasks

Core tasks cover fundamental operations that are common to most build and deployment processes. This includes tasks such as `<delete>`, `<copy>`, `<move>`, and `<tar>`.

Core tasks

Core tasks cover fundamental operations that are common to most build and deployment processes. This includes tasks such as `<delete>`, `<copy>`, `<move>`, and `<tar>`.

In general, optional tasks tend to be more specialized or specific to a software product, although this is not entirely the case.

Optional tasks

The optional tasks include items such as `<ftp>` and `<telnet>`. However, most optional tasks have to do with a specific software product, such as `<junit>`, or a procedure, such as EJB deployment.

Optional tasks

The optional tasks include items such as `<ftp>` and `<telnet>`. However, most optional tasks have to do with a specific software product, such as `<junit>`, or a procedure, such as EJB deployment.

Another note about optional tasks is that they are included in a separate `.jar` file (`optional.jar`) from the core tasks (`ant.jar`).

A skeletal buildfile (build3.xml)

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <project name="myproject" default="test.ant"
3         basedir=".">
4   <target name="test.ant"
5         description="A simple build file to test ant.">
6     <echo>Ant is working properly</echo>
7   </target>
8 </project>
```

```
[beetgreens]stg: ant -buildfile build3.xml
Buildfile: build3.xml

test.ant:
     [echo] Ant is working properly

BUILD SUCCESSFUL
Total time: 1 second
```

Running `ant -help` (extract)

```
ant [options] [target [target2 [target3] ...]]
Options:
  -help                  print this message
  -projecthelp           print project help information
  -version               print the version information
                           and exit
  -diagnostics           print information that might be
                           helpful to diagnose or report
                           problems.
  -quiet, -q             be extra quiet
  -verbose, -v           be extra verbose
  -debug                 print debugging information
  .....................................
```

Using an Ant Initialization File

Invoking Ant runs a wrapper script for a specific operating system, which starts a Java Virtual Machine (JVM) that runs the Ant Java code.

Using an Ant Initialization File

Invoking Ant runs a wrapper script for a specific operating system, which starts a Java Virtual Machine (JVM) that runs the Ant Java code.

Ant's behavior can be altered by setting environment variables that are passed from the wrapper script to the JVM. Environment variables can also be set in platform-dependent files, which are called by the Ant wrapper scripts.

When running Ant on a Unix-based platform, such as Linux, Solaris, Mac OS X, and Cygwin, the Unix wrapper script for Ant will look for the file `~/.antrc` before invoking Ant in the JVM.

When running Ant on a Unix-based platform, such as Linux, Solaris, Mac OS X, and Cygwin, the Unix wrapper script for Ant will look for the file `~/.antrc` before invoking Ant in the JVM.

The purpose of the environment variable `ANT_OPTS` is really to contain options to pass to the JVM.

```
[beetgreens]stg: export ANT_OPTS="-showversion"
[beetgreens]stg: ant -buildfile build3.xml
java version "1.4.2_02"
Java(TM) 2 Runtime Environment, Standard Edition (
Java HotSpot(TM) Client VM (build 1.4.2_02-b03, mi

Buildfile: build3.xml

test.ant:
     [echo] Ant is working properly

BUILD SUCCESSFUL
Total time: 1 second
```

Basic project management buildfile

```
1  <?xml version="1.0" ?>
2  <project name="AntTest" default="compile"
3        basedir=".">
4
5    <!-- compile target -->
6    <target name="compile"
7       description="Compile all of the source code.">
8      <javac srcdir="/home/stg/ant/tst" />
9    </target>
10
11 </project>
```

Running ant

```
[beetgreens]stg: ant
Buildfile: build.xml

compile:
    [javac] Compiling 7 source files

BUILD SUCCESSFUL
Total time: 20 seconds
```

Re-running ant

```
[beetgreens]stg: ant
Buildfile: build.xml

compile:

BUILD SUCCESSFUL
Total time: 2 seconds
```

Using properties in ant

```
1  <?xml version="1.0" ?>
2  <project name="AntTest" default="compile"
3        basedir=".">
4
5    <property name="dirs.source"
6       value="/home/stg/ant/tst">
7
8    <target name="compile"
9       description="Compile all of the source code.">
10     <javac srcdir="${dirs.source}" />
11   </target>
12
13 </project>
```

Example targets: making a TAR archive

```
1  <!-- backupTar target -->
2  <target name="backupTar"
3      description="Backs up all source into a tar file.">
4    <mkdir dir="${dirs.backup}" />
5
6    <tar  tarfile ="${dirs.backup}/${backupFile}"
7      basedir="${dirs.source}"
8      compression="gzip" />
9  </target>
```

Files can be timestamped using `<tstamp />`.

Target dependencies

Currently, each target in the buildfile is standalone in that it doesn't require another target to execute first. As other targets are added, the order in which targets are executed may become an issue.

Target dependencies

Currently, each target in the buildfile is standalone in that it doesn't require another target to execute first. As other targets are added, the order in which targets are executed may become an issue.

In our buildfile, a target will be added that will FTP the gzipped tarball to another server that gets backed up. We could create a target that replicates the `backupTar` target, and then does the FTP.

But this unnecessarily increases the size of the buildfile, and more importantly, it creates maintenance problems. If for some reason we wanted to create zip files instead of gzipped tarballs, we would have to change this in more than one place.

But this unnecessarily increases the size of the buildfile, and more importantly, it creates maintenance problems. If for some reason we wanted to create zip files instead of gzipped tarballs, we would have to change this in more than one place.

A better solution is to create a new target that does only the FTP step, and make it dependent on the `backupTar` target.

But this unnecessarily increases the size of the buildfile, and more importantly, it creates maintenance problems. If for some reason we wanted to create zip files instead of gzipped tarballs, we would have to change this in more than one place.

A better solution is to create a new target that does only the FTP step, and make it dependent on the `backupTar` target.

This way, if the `ftp` target is executed, it will first execute the `backupTar` target, and create the backup archive.

Any target can be defined as having a dependency on one or more other targets. By defining dependencies on targets, it's possible to ensure that all the targets in a project get executed in an acceptable order.

Dependencies in a buildfile

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <project name="targetDependencies"
4     default="tgt_1" basedir=".">
5
6    <target name="tgt_1" description="target 1">
7      <!-- tasks here -->
8    </target>
9
10   <target name="tgt_2"
11     depends="tgt_1" description="target 2">
12     <!-- tasks here -->
13   </target>
14
```

```
15    <target name="tgt_3"
16      description="target 3">
17      <!-- tasks here -->
18    </target>
19
20    <target name="tgt_4"
21      depends="tgt_3, tgt_2, tgt_1">
22      <!-- tasks here -->
23    </target>
24  </project>
```

Ant's behaviour

In this example, the final target is tgt_4. By default, Ant attempts to execute the target dependencies in the order listed in the target.

Ant's behaviour

In this example, the final target is tgt_4. By default, Ant attempts to execute the target dependencies in the order listed in the target.

If we were to execute a target with no dependencies, such as tgt_1, it simply executes.

Ant's behaviour

In this example, the final target is tgt_4. By default, Ant attempts to execute the target dependencies in the order listed in the target.

If we were to execute a target with no dependencies, such as tgt_1, it simply executes.

If we execute a target with a single dependency, tgt_2, Ant will determine that tgt_1 needs to execute first. So tgt_1 will execute followed by tgt_2.

In the case where there are multiple dependencies, as in tgt_4, Ant attempts to execute the targets from left to right in the dependency list.

In the case where there are multiple dependencies, as in tgt_4, Ant attempts to execute the targets from left to right in the dependency list.

Ant starts with tgt_3. Because tgt_3 has no other dependencies, it executes. Then Ant moves to tgt_2, but it has a dependency on tgt_1. So Ant will execute tgt_1, followed by tgt_2. Then Ant moves to the last target in the list, tgt_1. However, tgt_1 already executed, so Ant won't run it again.

```
[beetgreens]stg: ant -buildfile deps.xml tgt_4
Buildfile: deps.xml

tgt_3:

tgt_1:

tgt_2:

tgt_4:

BUILD SUCCESSFUL
Total time: 1 second
```

Optimising Ant build scripts

Sometimes it's tempting to daisy-chain a series of targets like compile, unittest, and javadoc, where unittest depends on compile, and javadoc depends on unittest.

Optimising Ant build scripts

Sometimes it's tempting to daisy-chain a series of targets like compile, unittest, and javadoc, where unittest depends on compile, and javadoc depends on unittest.

In fact, maybe javadoc only needs to depend on compile. Introducing unneeded dependencies only complicates matters unnecessarily. If a user wants to run javadoc, they shouldn't be forced to run unit tests unnecessarily. So use the minimum number of dependencies possible, and let Ant sort it out.

Entering a User ID/Password at Runtime

Ant provides the `<input>` task, which provides a means for prompting the user for additional information at runtime.

Entering a User ID/Password at Runtime

Ant provides the `<input>` task, which provides a means for prompting the user for additional information at runtime.

Here is an example of the use of the input task to prompt the user to enter their user id, and store their response in the property ftpUserID:

```
<input message="Please enter ftp user id:"
    addproperty="ftpUserID" />
```

By adding input tasks to the target, the user can be prompted to enter their user ID and password. Those entered values can then be stored in properties, and used in other tasks later in the buildfile.

By adding input tasks to the target, the user can be prompted to enter their user ID and password. Those entered values can then be stored in properties, and used in other tasks later in the buildfile.

What should Ant do if the user doesn't supply these values? In order to make the buildfile exit gracefully, the `<condition>` element can be used to check for the presence of an attribute.

Here's an example of the use of this element:

```
<condition property="noFTPUserID">
  <equals arg1="" arg2="${ftpUserID}" />
</condition>
```

What this does is check if the property ${ftpUserID} (arg2) is empty (arg1).

Here's an example of the use of this element:

```
<condition property="noFTPUserID">
  <equals arg1="" arg2="${ftpUserID}" />
</condition>
```

What this does is check if the property ${ftpUserID} (arg2) is empty (arg1).

If the `<equals>` condition is true, `<condition>` sets the property noFTPUserID.

Properties are immutable, and can only be set once. After this check is completed, a `<fail>` construct is encountered that looks like this:

```
<fail if="noFTPUserID">
    You did not enter your ftp user id.
</fail>
```

If the property noFTPUserID was set by the `<condition>` check, the build will print the message **You did not enter your ftp user id.**, and then exit gracefully.