Project Management using ProjMgr

Table of contents

1. Overview

ProjMgr is a tool for managing projects like XLattice which consist of a number of **components** which are used together but developed more or less independently. Each component will have its own source and test directories and the software for each will be delivered separately. However shared resources such as jars will be found in a common repository.

XLattice uses Ant to manage builds and JUnit to automate unit testing. All components inherit common properties from a shared `project.xml` control file and then have specific characteristics specified in a local `project.xml` . The Ant builds and JUnit unit tests are controlled by two files: `classpath.sh` sets up the Java classpath and `build.sh` is used by Ant during the build (under Windows, `.bat` files provide the same service).

ProjMgr helps keep component development consistent by generating build files from the two `project.xml` files. If executed in a component's base directory, ProjMgr will generate five files:

```
build.xml     -- the Ant build file
classpath.sh  -- sets up the Java classpath, invoked by build.sh
build.sh      -- runs Ant using build.xml
classpath.bat -- Windows equivalent of classpath.sh
build.bat     -- runs Ant under Windows
```

(Note: the Windows versions of these files are not created by `projmgr 0.1,` *but should be in the next release.)*

The `init` target of `build.xml` will make sure that all necessary component subdirectories exist. To create a new XLattice subproject, `dummy` , for example, requires two steps. First,

```
cd $XLATTICE_HOME
mkdir dummy
cd dummy
projmgr -a -c
```

creates the necessary subdirectory and configuration files. Then

```
./build.sh init
```

creates the subdirectories for Java source code, unit tests, documentation (`jml` and `xdocs`), and build results ('target' and its subdirectories).

2. Directory Structure

The current top-level XLattice directory structure is:

```
xlattice
```

```
  project.xml
  bin
  lib
  corexml
  projmgr
  util
```

The first file, `project.xml` , sets up configuration elements shared by the entire project.

2.1. bin

The `bin` directory must be on the path. In a UNIX/Linux environment running bash, this is effected by something similar to

```
export XLATTICE_HOME=$HOME/xlattice
export PATH=$XLATTICE_HOME/bin:$PATH
```

XLattice components normally put invocation scripts into `xlattice/bin` .

2.2. lib

The `lib` directory is a repository for the jars used by XLattice components. These are grouped into subdirectories by group ID, so that the XLattice jars, for example, are found in `xlattice/lib/xlattice` :

```
xlattice
    lib
      xlattice
        corexml-0.1.jar
        projmgr-0.1.jar
        util-0.1.jar
        LICENSE.txt
```

2.3. Component Subdirectories

`util, corexml,` and `projmgr` are the component subdirectories.

These have a common structure, with minor variations. For example, the `xlattice/corexml` subdirectory looks like this:

```
xlattice
  corexml
    project.xml
    build.xml
    classpath.bat
    classpath.sh
    build.bat
    build.sh
    src
      java
```

```
    test
  target
    classes
    test-classes
    docs
  jml
  xdocs
```

`project.xml` contains configuration information for the component.

The next five files are generated by `projmgr` from the configuration file, using both information from `project.xml` and defaults from the parent, `../project.xml` .

The directories that follow are created by Ant the first time it is run. `src/java` contains Java source code organized by package name. The code for `org.xlattice.projmgr` , for example, is found under `src/java/org/xlattice/projmgr` . Source code for unit tests is in a parallel hierarchy under `src/test` .

During the build, compilation generates the same tree structures under `target` , with source code classes under `target/classes` and compiled test clases under `target/test-classes` .

2.4. Documentation

Documentation production is similarly automated. Javadocs generated from the source code are written to `target/docs/api` . HTML documentation generated from `.xml` files under `xdocs` also appears under `target/docs` .

The only part of the process that is not currently automated is the translation of JML scripts into the XML-format files used to generate HTML. This problem should be addressed in the near future, probably by `projmgr-0.3` .

3. The Project File

There is a project file, `project.xml,` for each XLattice component. This is an XML file containing a single `<project>` element. Its most important subelements are

name	description	required?
extends	path to any parent configuration file	no
id	a single-word name for the component, conventionally in lower case	yes
name	a more descriptive name or phrase, conventionally several words in mixed uppper and lower case	yes

version	a decimal number with optional single-letter extension, for example 4.2 or 0.1a	yes
description	a paragraph or so describing the component	no
shortDescription	brief summary of the above	no
logo	logo for the component, usually in `.png` file format, conventionally found under `xdocs/images`	no
organization	described in more detail below	no, defaults
dependencies	described in more detail below	no

The `<organization>` element is used for generating documentation. It is specified in the parent `project.xml` , can be overridden in the component configuration file, but should not be.

name	**description**	**required?**
name	legal name of the organization, for use in license and copyright notices	no
url	URL of the organization's Web site	no
logo	organization logo, for example "xdocs/images/xlattice.png"	no

If there are any dependencies, they are described by repeating `<dependency>` subelements. These have the form

name	**description**	**required?**
groupId	group name, such as `xlattice` or `ant`	yes
artifactId	product or component name	yes
version	version number consisting of alphanumeric characters, digits, dots, dashes, but no spaces	no
type	file extension, defaults to `jar`	no
url	where to get the dependency	yes

These subelements are used to build a file name of the form `artifactId.type` or

`artifactId-version.type` , depending upon whether a version is specified. Ant will look for this in `libdir/groupId` . If it is not present, Ant will try to get it over the Internet from `urlartifactId-version.type` .

So if for example we have

groupId	xlattice
artifactId	util
version	0.1
url	http://www.xlattice.org/jars/

then Ant will look for `../lib/xlattice/util-0.1.jar` and failing to find that will attempt to fetch it from `http://www.xlattice.org/jars/util-0.1.jar` .

Notice that while some punctuation marks are automatically supplied, the terminating slash on the URL cannot be. If there is no slash at the end of the URL, the system interprets this as an instruction to add a space to the URL before appending the name of the jar.

A future revision of `projmgr` will change the structure of the `dependency` element, probably by adding an optional or alternative `fullurl` subelement, possibly in other ways.

4. Build File Generation using ProjMgr

`projmgr` is a command line utility. In its current form the UNIX version takes several options:

```
projmgr [-a] [-c] [-h] [-v]

  -a  generate build.xml for Ant
  -c  create the classpath.{bat,sh} and build.{bat, sh} command files
  -h  display a help message
  -v  show the version number
```

If no option is specified, running `projmgr` has no effect.

Configuration and command files are created in the current directory from the information in `./project.xml` .

5. JML

XLattice currently uses Maven to generate its HTML documentation. Maven expects input as XML, in so-called *Anakia* format.

Note:

These references to Maven are obsolete.

Writing in XML is tedious, error-prone, and time-consuming. JML, part of the ProjMgr component, makes the author's job much simpler by automatically generating XML from ordinary text with some lightweight markups.

JML is based on the Antlr lexer/parser generator and was specifically inspired by Terence Parr's TML . It differs from TML in minor points of syntax but more importantly in what it generates. Whereas TML converts text directly into HTML, JML produces XML in a form designed for further transformation. This allows us to use JML to produce body copy like this, but then feed the output into other tools which add menus, headers, and footers to make the finished HTML Web page.

JML is more fully documented in the ProjMgr API, which can be reached via the menu to the left. An Ant task is also forthcoming. On the other hand, many will find it easier to understand how to use JML by looking through the JML script that produced this page, `jml/components/projmgr/index.jml`.