**Using MsBuild**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data** | **Changes** | **Reviser** |  |
| 2010.11.04 | Initial version | Jurgen |  |
|  |  |  |  |

# Purpose 🡺 Software project-, building- and release management.

This document serves as a technical roadmap to implement Software Project, Configuration, Release and Dependency management at Virtuos.

## Default Lifecycle

|  |  |
| --- | --- |
| validate | validate the project is correct and all necessary information is available. |
| initialize | initialize build state, e.g. set properties or create directories. |
| generate-sources | generate any source code for inclusion in compilation. |
| process-sources | process the source code, for example to filter any values. |
| generate-resources | generate resources for inclusion in the package. |
| process-resources | copy and process the resources into the destination directory, ready for packaging. |
| compile | compile the source code of the project. |
| post-process-compile | post-process the generated files from compilation. |
| generate-test-sources | generate any test source code for inclusion in compilation. |
| process-test-sources | process the test source code, for example to filter any values. |
| generate-test-resources | create resources for testing. |
| process-test-resources | copy and process the resources into the test destination directory. |
| test-compile | compile the test source code into the test destination directory |
| post-process-test-compile | post-process the generated files from test compilation. |
| test-run | run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed. |
| prepare-package | perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package. |
| package | take the compiled code and package it in its distributable format, such as a JAR. |
| pre-integration-test | perform actions required before integration tests are executed. This may involve things such as setting up the required environment. |
| integration-test-run | process and deploy the package if necessary into an environment where integration tests can be run. |
| post-integration-test | perform actions required after integration tests have been executed. This may include cleaning up the environment. |
| verify | run any checks to verify the package is valid and meets quality criteria. |
| install | install the package into the local repository, for use as a dependency in other projects locally. |
| deploy | done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects. |

## A Build Lifecycle is Made-up of Phases

Each build lifecycle is defined by a list of build phases, wherein a build phase represents a stage in the lifecycle. For example, the default lifecycle has the following build phases:

* Validate
* Compile
* Test
* Package
* Integration-Test
* Verify
* Install
* Deploy

These build phases (plus the other build phases not shown here) are executed sequentially to complete the default lifecycle. Given the build phases above, this means that when the default lifecycle is used, it will first validate the project, then will try to compile the sources, run those against the tests, package the binaries, run integration tests against that package, verify the package, install the verified package to the local repository, then deploy the installed package in a specified environment.

To do all those, you only need to call the last build phase to be executed, in this case, deploy:

mvn deploy

That is because if you call a build phase, it will execute not only that build phase, but also every build phase prior to the called build phase. Thus, doing

mvn integration-test

will do every build phase before it (validate, compile, package, etc.), before executing integration-test.

There are more commands that are part of the lifecycle, which will be discussed in the following sections.

It should also be noted that the same command can be used in a multi-module scenario (i.e. a project with one or more subprojects). For example:

mvn clean install

This command will traverse into all of the subprojects and run clean, then install (including all of the prior steps).

## The features:

* One build tool (MsBuild)
* One uniform convention for all projects
* Quick project setup (construct)
* Configuration management (versioning, deploy and install)
* Dependency management
* Release management (deploy)

## The goals:

* To reduce waste (not able to re-use, project setup, continues integration setup, build process, build time)
* To promote standards and best practices
* To promote re-use of work in the form of libraries
* To enable distributed code compiling for all platforms consistently to minimize building time

## How To

Using MsBuild we can accomplish all of the above and much more. At this moment in time Microsoft has released Visual Studio 2010, which for C++ has introduced a way to fully control the build process. The building process in Visual Studio 2010 is using MsBuild 4.0 and it allows you to customize the UI in Visual Studio as well as the customizing the build process.

In other words, we can implement new compilers, tools and tasks as we see fit.

## Build Management

This deals with how builds are created for different platforms, using which configuration.

Currently we have these variables:

* Target: Debug, Release, Profile and Final
* Type: Dev, Client, Retail
* Territory: EU, US, JP

## Release Management

This deals with how builds are deployed as well as installed.

We need a local package cache, $(SCM\_LOCAL\_REPO\_ROOT).

## Dependency Management

This deals with how the build process determines dependencies and its relation with Release Management. A convention on how a project and package describe the dependencies is also part of this.

## Configuration Management

This deals with how builds are named and versioned and how dependencies are declared.

For naming and versioning of a package we will use the following convention:

PACKAGE\_NAME-VERSION-PLATFORM.ZIP (e.g.: xmath-1.0.2010.07.01-PS3.zip)

Libraries inside will have the following naming:

PACKAGE\_NAME-TARGET-TYPE-PROCESSOR-TERRITORY (e.g.: xmath-Debug-Dev-PPU-WW.lib)

VERSION: 1.0.2010.07.01 (Major.Minor.Year.Month.Day)

TARGET: Debug, Release, Profile, Final

TYPE: Dev, Client, Retail

PROCESSOR: Native, x32, x64, PPU, SPU

PLATFORM: DS, DSI, 3DS, WII, PSP, PS2, PS3, 360, PC

TERRITORY: WW, EU, US, JP

# The high level tasks

## Construct

This is the task that will generate the folder structure according to its convention. It may also generate some files and anything necessary to setup a project.

## Clean

This is the task that will remove and temporary and generated files from the project.

## Verify

This will verify if the package server is accessible, if all the package dependencies are locally and remotely available and if the environment is ok (Windows version, .NET version, Intel/AMD processor, etc.)

## Build

This will build the main package which can be an application or library (a static .lib or a dynamic .dll).

## Version Inc

Increments the version of this package

## Test Build

This will build all the unit tests for all the platforms and configurations that this project supports.

## Test Run

This will run all necessary unit tests for all the platforms that this project supports.

## Deploy

Deploy is responsible for versioning and submitting a build and tested package to the package server.

## Install

This will install all defined package dependencies as well as the necessary packages of the dependencies of the installed packages.

## Construct

Here we need a predefined (configurable) directory structure, we have agreed on the following:

|  |
| --- |
| **/[package name]**  **/source**  **/main**  **/resources**  **/include/[package name]**  **/cpp, /cs**  **/test**  **/resources**  **/cpp, /cs**  **/target**  **/documents**  **/manuals**  **/tools** |

## Install

The installation of a package will first cache the package as a .zip file in the package cache. After that the package is extracted to the target folder of the package that has this dependency. The extracted package will look like this in the target folder:

|  |
| --- |
| **/[package name]**  **/source**  **/main**  **/resources**  **/include/[package name]**  **/cpp, /cs**  **/test**  **/resources**  **/cpp, /cs**  **/documents**  **/manuals**  **/tools**  **/target**  **/[package name] {dependency}**  **/include**  **/[package name]**  **/lib**  **/[package name]**  **/documents**  **/manuals**  **/references**  **/tools** |

# Ultimate Goal

Note: I have installed our MsBuild package

I go to my project folder and type

**msbuild scm.proj /t:Construct /p:language=c++ /p:platform=PC /p:name=MyProject**

I navigate into the MyProject folder and see 2 files, MyProject.props and MyProject.targets and start to edit MyProject.props and add platforms, configurations and dependencies.

**msbuild scm.proj /t:Construct**

This will create the folder layout as well as visual C++ projects in src/main/cpp and src/test/cpp called package\_name.vcxproj and package\_name\_test.vcxproj. The generated c++ project files will contain the default configurations Debug, Release, Profile and Final for Dev, Client and Retail.

**Dev\_Debug, Client\_Debug, Retail\_Debug**

**Dev\_Release, Client\_Release, Retail\_Release**

**Dev\_Profile, Client\_Profile, Retail\_Profile**

**Dev\_Final, Client\_Final, Retail\_Final**

It will also add PREPROCESSOR DEFINES according to our convention.

I am running Microsoft Visual Studio 2010; I open the .vcxproj project in MyProject/src/main/cpp and can see the configuration and platforms. When opening the project properties I can see the PREPROCESSOR DEFINES that have been prepared and I can also see the Debug, Release, Profile and Final settings for every platform have been set to their preferred defaults. I can also see the additional libraries that the linker should link with.