Checklist for Test Tool release

Version 2.0p

Michael Stahl, 7-Nov-2016

michael.stahl@intel.com / [www.testprincipia.com](http://www.testprincipia.com)

**Notes:**

The decision if a checklist item is mandatory or not is up to the adopting organization.

Items marked with a red star are my personal recommendation for absolutely minimum mandatory items. What I really recommend is to work towards adopting all the items on this checklist.

My work environment is Windows, with Visual Studio as the main development tool. The rules below are influenced by it. If you work in a different environment, consider editing this list to better fit your case.

# Tools Rules

1. \*Decide the tool’s name and use exactly this name everywhere you refer to the tool (Visual Studio solution name, main source code file name, folder names, documentation, every-day conversations). Make it unique and comfortable to use. Don’t call all your tools “TestTool” or “AutoTest”….
	* 1. Rationale: Without unique names you now need to identify tools by description (“use my AutoTest… don’t use John’s; use the csv file AnalysisTool – not the Excel AnalysisTool…). It also gives everyone a chance to be creative and immortalize themselves. Select a name that means something to you; holds your initials or part of your name… be creative. Or at least make up a catchy, descriptive name. It’s fun.
2. \*Avoid using any hard-coded, specific file names or directory paths inside the code. Use macro variables (e.g. $(PROJET\_DIR) ) or relative paths ( “..\..\” etc. )
3. \*Load the tool’s code to your configuration management (CM) system. DO NOT wait till the tool is ready. Start on day one of development. The folder name in the CM system should be the official tool name.
4. \*Continuously check-in changes to avoid work loss in case of hard disk crash.
5. \*The tool should have a minimal “help” information available to users. Either via a “help -> About” or via “ --help” switch. The format should be like you’d expect such a command to show: general call structure, options, switches and a short (one sentence) explanation on each.

For tools that accept inputs from the command line, have the Help blurb print when no input is given or when too few or too many input items are given – not only when the user calls for --help.

1. \*Add a Version to the tool.
	1. For C++ tools, see [here](http://testprincipia.com/other-testing-stuff/#stuff4) how to do it (open the ZIP file and read the Readme.docx file).
	2. For C# tools, see [here](http://stackoverflow.com/questions/356543/can-i-automatically-increment-the-file-build-version-when-using-visual-studio) how to do it
	3. For Python or any other .exe files, see [Appendix A](#_Appendix_A_–) how to do it
2. Clean the code (remove redundant, commented-out lines; do indentation; add empty lines to increase readability)
3. Add comments to the code. Especially where you did things that are not obvious. Be very verbal. You will be the first to enjoy a reminder what the code does when you need to maintain it in six months’ time.
4. Add a comment at the top of the main code file, explain what this tool is all about and add the data and your name there. Also (important!) explain what pre-requisites and dependencies are needed to allow compiling the code (e.g.: QT, Visual Studio runtime, etc. Indicate the versions of the dependencies).
	1. If you have only very few dependencies (e.g. two dll files) consider just adding them to the solution, instead of forcing the next person to install a whole package to be able to compile or use the tool.
5. If you copy-pasted a good chunk of code from the internet, add the URL to the code you used and mark or explain what was copied, if and how it was changed. This directive does not apply to short solutions copied from stack exchange and such. More relevant to full algorithms or complete tutorial code. When in doubt – add the URL.
	1. Rationale: if in the future your company decides to publish your tool to customers as a test tool, there are legal implications (copyright aspects) that will need to be resolved. Knowing the source of the code will make life easier.
6. Add tests to the tool. These may be tests written in GoogleTest and added as a project to the tool’s solution. They can also be any other types of tests (e.g. a batch file that runs the tool, and compare the results to some golden version of the results). These tests must pass before a new version of the tool is released.
	1. The tests must be checked in together with the tool, to the configuration management (CM) system.
	2. This item in the checklist is very time consuming. My experience is that it takes at least as much time as writing the tool. Yet it is critical. You are going to make pass/fail decisions based on this SW tool you are writing. And SW tends to be buggy…
7. Perform code review with someone in your team.
	1. Teams should define who does code review with whom
	2. This item is one of the hardest ones in the checklist since it calls for effort by someone else other than the tool writer. It make sense to postpone it as mandatory till the other, more technically straight-forward rules becomes ingrained as part of your process.
8. \*Add a label in the CM system so that all files used to build a specific tool version can be easily obtained in the future. A recommended label text: <Tool Name> version <version number> . Example: “myTool.exe version 1.0.0.5”
9. \*Place the tool in a network location that is your team’s “tools location”. The tool and all the needed support files need to be in a folder whose name is exactly as the tool’s name.

The tool’s folder must include:

* 1. The tool’s .exe or installer
	2. Any needed dependencies (libraries; dlls, redistributables etc.). Put ONLY the needed dependencies. Don’t just copy a whole bunch of dll’s from a large package you installed, of which you are using only one dll.
	3. A short Readme.txt file explaining what this tool does.
		1. Can be a copy of the “help” blurb that the code would show the user when started with “--help” switch or in the “Help->about” window.
		2. State clearly – right at the start, after the version info, if the tool has any pre-requisites and dependencies.
		3. Maintain a version history, where you give some details what changed in the new version. This will tell people who used the tool in the past if they have to re-read the ReadMe.
	4. An example that works. One that the users can take to see how the tool works. Note that many times the tests you wrote in step 11 can serve as examples.

Any file that is in the tool’s folder, must also be added to the CM system.

1. If you built an organization-wide list of internally-developed tools, add the tool to that list.
2. Get a clean machine (one that does not have any pre-requisites installed on it) – just a clear OS install. Copy the tool to this new machine, install whatever pre-requisites are indicated in the ReadMe.txt and try to run it. It should run cleanly.
3. Install whatever IDE is needed to compile the tool. Install the pre-requisites listed at the top of the main code file. Download the code from the CM system and try to compile. It should compile cleanly.

# Checklist

* \*Pick a name for the tool
* \*No hard-wired filenames or paths
* \*Tool in CM
* \*Latest changes are checked in
* \*Minimal “help” information available
* \*Added Version to the tool
* Code is cleaned up
* Redundant files removed
* Comments added to the code
* General comment at the top of the main code file (including compile pre-requisites)
* Code from the web is marked, and a URL to the source exists
* Tests added to the tool, added to the CM system and are passing
* Code review is done and comments resolved
* Add Label in the CM system
* \*Tool in the central tool location for your team
* .exe or installer
* Any needed dependencies
* Readme.txt file
* An example that works
* Support files in the tool’s folder added also to the CM system
* Tool listed on the organization’s internal tools list
* The tool runs correctly on a clean machine
* The tool compile cleanly on a clean machine

# Appendix A: Adding version to Python script and to any .exe file

Warning! This is a kludge. But it works. From step 3 it works for any .exe on a Windows system.

Create an exe file from your Python script:

1. Install pyInstaller using

pip install pyinstaller

(this installation worked on my system where I have Python 3.4.3 installed)

1. Run “pyinstaller --onefile <your python script>” to create an executable from your python scripts. The executable will be in a \dist folder in the same folder where your python script is.

Upversion the .exe:

1. Get **verpatch.exe** from: <http://www.codeproject.com/Articles/37133/Simple-Version-Resource-Tool-for-Windows> and place it somewhere that is in your PATH variable (so you can call this executable from any command shell without specifying the full path).
2. Get ExeUpversion.exe from [here](http://testprincipia.com/other-testing-stuff/#stuff4). Alternatively, create it by compiling the python code in [B.1](#_B.1_PyUpversion.py) (follow steps 1,2 above). Place it somewhere that is in your PATH.
3. The zip file for ExeUpversion contains also the ExeUpversion.bat file and an example how to use it. Alternatively, copy the script in [B.2](#_B.2_–_PyUpversion.bat) to a windows batch file called ExeUpversion.bat.
4. If the .exe file you want to up-version does not yet have a version number, create one using:

verpatch <filename> /va <x.y.z.w>

1. Run the ExeUpversion.bat batch file by dragging and dropping the exe on the ExeUpversion.bat file (or by running from a command shell:

ExeUpversion.bat <exe file name>

## B.1 ExeUpversion.py

# ExeUpversion

# ver 1.0.0.1 1/Oct/2016

# Author: Michael Stahl

#

# A very simple utility to help auto-increment version number on any executable.

#

# The utility relies on verpatch.exe (from: http://www.codeproject.com/Articles/37133/Simple-Version-Resource-Tool-for-Windows ) to do the actual work.

# All that this utility does is to get the seconds and days since the python epoch and use them to be the 2 minor numbers of the version of the executable.

# What this also means that it assumes you have already used verpatch.exe to (a) add a version to your executable and (b) the two higher version numbers are to your liking (if not, use verpatch to update these before using ExeUpversion).

#

#

import time, math, os, sys

if (len(sys.argv) < 2):

 print ("\n\nUsage: ExeUpversion <exe file name>")

 print ("\nExeUpversion will increment the last two digits of the input file's version number.")

 print ("Assumptions:\n - The file already has a version number\n - the two first digits are what you want them to be")

 print ("\nUse verpatch.exe from http://www.codeproject.com/Articles/37133/Simple-Version-Resource-Tool-for-Windows to create the first version and set the first two digits\n\n")

 sys.exit()

filename = sys.argv[1]

seconds = round(time.time())

days = math.floor(seconds / 86400)

# must divide by 2 since the number of seconds in a day is > 64K, and verpatch.exe which we use next to change the version, rolls-over the seconds number when it reaches 64K.

sec = math.floor((seconds - days\*86400)/2)

command\_string = "verpatch " + "\"" + filename + "\" " + str(days) + "." + str(sec)

print (command\_string)

os.system(command\_string)

## B.2 ExeUpversion.bat

@echo off

REM print the current version

echo The version before up-versioning:

verpatch "%~f1" /vo | find "FILEVERSION"

echo.

REM update the version

..\ExeUpversion "%~f1"

REM show the new version

echo.

echo The version after up-versioning:

verpatch "%~f1" /vo | find "FILEVERSION"

pause