|  |  |  |  |
| --- | --- | --- | --- |
| **Data** | **Changes** | **Reviser** |  |
| 2010.12.07 | Initial version | Jurgen |  |
|  |  |  |  |

# Purpose 🡺 Standard versioning, Version comparing and Version Range Specification

## Convention

Package versioning should be based on the Major.Minor.Fix.Year.Month.Day.Hour.Minute (1.0.0.0.12.11.14.54) scheme:

* *X indicate a change in the public API that introduce backward incompatibility*
* *Y indicate an addition of some features*
* *Z indicate a fix (either fixing a bug, either changing internal structure without impacting functionality)*
* *B indicates the build number and this will be automatically increased*

Required delimiters: ‘.’, ‘-‘

Characters Allowed: ‘0’-‘9’, ‘a’-‘z’, ‘A’-‘Z’

## Versioning and Database

How to be able to rebuild all versions when the database is lost?

* Tag all Deploy releases

What to store in the versioning database?

* Package[] (table)
  + Platform (string):
  + Branch (string): default
  + Version (string): Major.Minor.Fix.Year.Month.Day.Hour.Minute
  + VCS Type (string): Mercurial, Git, Bazaar, Perforce, AccuRev
  + VCS URL (string):
  + Revision (string):

Requirements:

* Concurrency
* Query
* Incorruptible

Solution:

Maybe using Mercurial is enough if we make sure that versioning never conflicts. If we use ‘hg tag’ it might happen that 2 developers work bump the ‘major’, ‘minor’ to the same number and this will conflict. We can reduce the conflict by tracking a version per platform, then the tagging needs to be like Platform.Major.Minor.Fix. Now whenever there is a conflict we can refuse to push it and ultimately deploy the new package. The only way for the developer to fix this is to verify and correct the Major.Minor.Fix and deploy again.

Using Mercurial we can now get the list of public versions (deploy), for the local versions (install) we need to introduce a build number, so the local versioning becomes Platform.Major.Minor.Fix.Build. The build number is reset once a Deploy is executed.

## Features

* Mixing of '-' (dash) and '.' (dot) separators
* Transition between characters and digits also constitutes a separator:
  + 1.0alpha1 => [1, 0, alpha, 1]; This fixes '1.0alpha10 < 1.0alpha2'
* Unlimited number of version components
* Version components in the text can be digits or strings
* Strings are checked for well-known qualifiers and the qualifier ordering is used for version ordering
  + well-known qualifiers (case insensitive)
    - snapshot (NOTE; snapshot needs discussion)
    - alpha or a
    - beta or b
    - milestone or m
    - rc or cr (Release Candidate)
    - ga or final (Gold Master) or ‘the empty string’
    - sp (Service Pack)
* Version components prefixed with '-' will result in a sub-list of version components. A dash usually precedes a qualifier, and is always less important than something preceeded with a dot. We need to somehow record the separators themselves, which is done by sublists.   
  Parse examples:  
  1.0-alpha1 => [1, 0, ["alpha", 1]]  
  1.0-rc-2 => [1, 0, ["rc", [2]]]

## Parsing versions

The version string is examined one character at a time. There is a buffer containing the current text - all characters are appended, except for '.' and '-'. Below, when it's stated 'append buffer to list', the buffer is first converted to an Integer item if that's possible, otherwise left alone as a String. It will only be appended if its length is not 0.

* If a '.' is encountered, the current buffer is appended to the current list, either as a Integer Item (if it's a number) or a String Item.
* If a '-' is encountered, do the same as when a '.' is encountered, then create a new sublist, append it to the current list and replace the current list with the new sub-list.
* If the last character was a digit:
  + and the current one is too, append it to the buffer.
  + otherwise append the current buffer to the list, reset the buffer with the current char as content
* If the last character was NOT a digit:
  + if the last character was also NOT a digit, append it to the buffer
  + if it is a digit, append buffer to list, set buffers content to the digit
* Finally, append the buffer to the list

Some examples:

* 1.0 => [1, 0]
* 1.0.1 => [1, 0, 1]
* 1-SNAPSHOT => [1, ["SNAPSHOT"]]
* 1-alpha10-SNAPSHOT => [1, ["alpha", "10", ["SNAPSHOT"]]]

## Ordering algorithm

Internally 3 version component types are used:

1. integer (IntegerItem)
2. string (StringItem) (knows if it's a qualifier or not)
3. sub list (ListItem)

Elements from both versions are compared one at a time; first the first element of both, then the second, etc…  
(Note: 'item' and 'component' are used interchangeably)

Table: Ordering rules when comparing version components

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Integer** | **String** | **List** | **Null** |
| **Integer** | Highest is newer | Integer is newer | Integer is newer | If Integer==0 then equal, otherwise integer is newer |
| **String** | Integer is newer | Order by well-known qualifiers and lexically (see below) | List is newer | Compare with “” |
| **List** | Integer is newer | List is newer | Version itself is a list; compare item by item | Compare with empty list item (recursion) this will finally result in String==?null or Integer==?null |
| **null** | If Integer==0 then equal, otherwise integer is newer | Compare with “” | Compare with empty list item (recursion) this will finally result in String==?null or Integer==?null | Doesn’t happen |

Special note on string comparing:

A predefined list of well-known qualifiers is present. For comparison, the string is converted to another string, as follows:

* First, the well-known qualifier list is consulted for presence of the string
* If the string is present, the index in the list is returned, as a string
* If the string is not present, then qualifiers.Count + "-" + string is returned.

Then the strings are lexically compared.  
Examples:

* "alpha" yields "1"
* "" yields "4"
* "abc" yields "7-abc"
* "xyz" yields "7-xyz"

String Compare examples:

* 1.0 ==? 1.0-alpha: "" (or null) ==? "alpha" -> "4" ==? "1" -> 1.0 is newer
* 1 ==? 1.0: equal
* 1-beta ==? 1-xyz: "2" ==? "7-xyz" -> 1-xyz is newer

Some comparisons that yield different results from the current implementation:

* 1-beta ==? 1-abc: "2" ==? "7-abc" -> 1-abc is newer
* 1.0 ==? 1.0-abc: "4" ==? "7-abc" -> 1.0-abc is newer
* 1.0-alpha-10 ==? 1.0-alpha-2: 10 > 2, so '1.0-alpha-10' is newer
* 1.0-alpha-1.0 ==? 1.0-alpha-1: equal
* 1.0-alpha-1.2 ==? 1.0-alpha-2: 1.0-alpha-2 is newer

## Version Range Specification

|  |  |
| --- | --- |
| Range | Meaning |
| 1.0 | x >= 1.0 |
| (,1.0] | x <= 1.0 |
| (,1.0) | x < 1.0 |
| [1.0] | x == 1.0 |
| [1.0,) | x >= 1.0 |
| (1.0,) | x > 1.0 |
| (1.0,2.0) | 1.0 < x < 2.0 |
| [1.0,2.0] | 1.0 <= x <= 2.0 |
| (,1.0],[1.2,) | x <= 1.0 or x >= 1.2. Multiple sets are comma-separated |
| (,1.1),(1.1,) | x != 1.1 |

Note: The use of [1.0,],[,2.0] is not recommended to specify a bounded inclusive range, for this the [1.0,2.0] notation should be used.