

Versioning

Improve default support for version schemes

The [current implementation for version schemes](#) is rather limited. It only supports 5 properties:

1. Major version
2. Minor version
3. Incremental version (bugfix)
4. Build number
5. A Qualifier.

Flaws

Other than the limitation of supported versions, the current implementation has several flaws:

- It only supports the following schemes:
 - 'positiveInteger-buildnumber' where buildnumber doesn't start with '0' and has to be
 - 'positiveInteger-qualifier'
 - positiveInteger(.positiveInteger(.positiveInteger))-(buildNr|qualifier)
- Inconsistent/unintuitive parsing:
 - in something-X, where X is [1..9], X will be a buildnumber
 - in something-0X, where X is any string, '0X' will be a qualifier
 - something.0something will yield 0.0.0.0-something.0something
 - something.NaN will also yield 0.0.0.0-something.NaN
- getBuildNumber returns '0' when no buildnumber is specified, yet you can never specify 0 as a buildnumber
- qualifiers are sorted lexically
- if a qualifier is a prefix of another, the shorter one is considered newer (example: '1.0-alpha10' is considered older than '1.0-alpha1')

The [unit tests that are there to test comparison](#) only check for a few cases with snapshots. When all tests (version A < version B) are expanded to also test for SNAPSHOTs (version A-SNAPSHOT < version B-SNAPSHOT)

a lot of the tests fail.

Left	Op	Right		Left	Op	Right	
1	=	1	✓	1-SNAPSHOT	=	1-SNAPSHOT	✓
1	<	2	✓	1-SNAPSHOT	<	2-SNAPSHOT	✓
1.5	<	2	✓	1.5-SNAPSHOT	<	2-SNAPSHOT	✓
1	<	2.5	✓	1-SNAPSHOT	<	2.5-SNAPSHOT	✓
1	=	1.0	✓	1-SNAPSHOT	=	1.0-SNAPSHOT	✓

1	=	1.0.0	✓	1-SNAPSHOT	=	1.0.0-SNAPSHOT	✓
1.0	<	1.1	✓	1.0-SNAPSHOT	<	1.1-SNAPSHOT	✓
1.1	<	1.2	✓	1.1-SNAPSHOT	<	1.2-SNAPSHOT	✓
1.0.0	<	1.1	✓	1.0.0-SNAPSHOT	<	1.1-SNAPSHOT	✓
1.1	<	1.2.0	✓	1.1-SNAPSHOT	<	1.2.0-SNAPSHOT	✓
1.0-alpha-1	<	1.0	✓	1.0-alpha-1-SNAPSHOT	<	1.0-SNAPSHOT	✗
1.0-alpha-1	<	1.0-alpha-2	✓	1.0-alpha-1-SNAPSHOT	<	1.0-alpha-2-SNAPSHOT	✓
1.0-alpha-1	<	1.0-beta-1	✓	1.0-alpha-1-SNAPSHOT	<	1.0-beta-1-SNAPSHOT	✓
1.0	<	1.0-1	✓	1.0-SNAPSHOT	<	1.0-1-SNAPSHOT	✗
1.0-1	<	1.0-2	✓	1.0-1-SNAPSHOT	<	1.0-2-SNAPSHOT	✗
2.0-0	=	2.0	✓	2.0-0-SNAPSHOT	=	2.0-SNAPSHOT	✗
2.0	<	2.0-1	✓	2.0-SNAPSHOT	<	2.0-1-SNAPSHOT	✗
2.0.0	<	2.0-1	✓	2.0.0-SNAPSHOT	<	2.0-1-SNAPSHOT	✗
2.0-1	<	2.0.1	✓	2.0-1-SNAPSHOT	<	2.0.1-SNAPSHOT	✓
2.0.1-klm	<	2.0.1-lmn	✓	2.0.1-klm-SNAPSHOT	<	2.0.1-lmn-SNAPSHOT	✓

2.0.1-xyz	<	2.0.1	✓	2.0.1-xyz-S NAPSHOT	<	2.0.1-SNAP SHOT	✗
2.0.1	<	2.0.1-123	✓	2.0.1-SNAP SHOT	<	2.0.1-123-S NAPSHOT	✗
2.0.1-xyz	<	2.0.1-123	✓	2.0.1-xyz-S NAPSHOT	<	2.0.1-123-S NAPSHOT	✗

Proposal

I'm proposing the following implementation: [GenericArtifactVersion.java](#) (unit test: [GenericArtifactVersionTest.java](#)). **It has been integrated in artifact 3.0-SNAPSHOT r656775(15/5/2008) as [ComparableVersion.java](#).**

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
 - (the empty string) or ga or final
 - sp
- 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 preceded 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's 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 it's length is not 0.

- If a '.' is encountered, the current buffer is appended to the current list, either as a IntegerItem (if it's a number) or a StringItem.
- 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:

- integer (IntegerItem)
- string (StringItem) (knows if it's a qualifier or not)
- sublist (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)

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.size() + "-" + 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`

Note: This approach differs from the version comparison as done by OSGi [\[0\]](#).

Make version handling pluggable

When somebody devises a version scheme that cannot be handled by the above, it should be possible to plug in a new scheme. Two possible scenarios for unsupported schemes:

- The dash hash higher priority than the dot: 1-0 is newer than 1.0
- Dashes and dots have the same priority: 1.1 == 1-1.
- Qualifier order is different, or unknown qualifiers.

To make version schemes pluggable, the following is required:

- A POM change to support something like this to identify a version-scheme implementation artifact:

```
<versionScheme>
  <groupId>..</groupId>
  <artifactId>..</artifactId>
  <version>..</version> <!\-\- we may need to disallow version ranges
here \-->
</versionScheme>
```

- Maven-metadata at the artifact level needs to include the tag above. We'll limit version schemes on a per artifact basis. This is required in order to resolve versions using ranges.
- An interface definition for `VersionScheme`
- A way to detect what is the version class inside the version-scheme artifact; I hope we can use plexus, as

long as multiple version-scheme implementations (same hint, same package/classname) can be accessed simultaneously without conflict.

- Refactoring the version code out of maven-artifact so plugin code etc. can use it too
- The super pom will contain a default versionScheme tag listing the maven internal implementation

Define a grammar for version specifications

I'm not entirely sure this is necessary, but for other languages that cannot use pre-packaged version scheme implementations in Java, we need to have some sort of metadata, preferably in the version-scheme artifact, describing the version scheme. Perhaps something like
version-scheme-1.0.jar!/META-INF/maven/version-scheme.file-extension

Several proposals have been made for the version scheme description language:

- regular expressions. Downside: you cannot record version ordering information, only parse rules
- (E)BNF grammar. Same downside, though the operator ('.' and '-') priority can be expressed by the level in the AST.
- XSD.
- XML, where we provide an XSD with the grammar for the grammar.

As an example here's an XSD you could use to describe versions:

```
<xs:schema>
  <xs:element
name="versionSchemeDefinition">
  <xs:complexType>
    <xs:sequence>
      <!-- the order of
qualifierDefinitions is from oldest to
newest -->
      <xs:element
ref="qualifierDefinition" minOccurs="0"
maxOccurs="unbounded"/>
      <xs:choice maxOccurs="unbounded"
minOccurs="0">
        <xs:element
ref="stringComponent"/>
        <xs:element
ref="numberComponent"/>
        <xs:element ref="subComponent"/>

```

```
        </xs:choice>
    </xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="qualifierDefinition">
    <xs:complexType>
        <xs:attribute name="name"
type="xs:string"/>
        <xs:attribute name="caseSensitive"
type="xs:boolean" default="false"/>
    </xs:complexType>
</xs:element>

<xs:element name="stringComponent">
    <xs:complexType>
        <xs:attribute name="name"
type="xs:string"/>
        <xs:attribute name="prefix"
type="xs:string" default="."/>
    </xs:complexType>
</xs:element>

<xs:element name="numberComponent"
type="xs:int">
    <xs:complexType>
        <xs:attribute name="name"
type="xs:string"/>
        <xs:attribute name="prefix"
type="xs:string" default="."/>
        <xs:attribute name="optional"
```

```
type="xs:boolean" default="false"/>
  </xs:complexType>
</xs:element>

<xs:element name="subComponent">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded"
minOccurs="0">
      <xs:element ref="stringComponent"/>
      <xs:element ref="numberComponent"/>
      <xs:element ref="subComponent"/>
    </xs:choice>
    <xs:attribute name="name"
type="xs:string"/>
    <xs:attribute name="prefix"
type="xs:string" default="-"/>
  </xs:complexType>
```

```
</xs:element>
</xs:schema>
```

A sample scheme:

```
<versionSchemeDefinition>
  <qualifierDefinition name="snapshot"/>
  <qualifierDefinition name="alpha"/>
  <qualifierDefinition name="beta"/>
  <qualifierDefinition name="rc"/>
  <qualifierDefinition name=""/>
  <qualifierDefinition name="ga"/>
  <qualifierDefinition name="sp"/>
  <numberComponent name="major"/>
  <numberComponent name="minor"
optional="true"/>
  <numberComponent name="micro"
optional="true"/>
  <stringComponent name="qualifier"
optional="true"/>
  <numberComponent name="buildnumber"
optional="true"/>
</versionSchemeDefinition>
```

This scheme doesn't even come close to being able to describe the variety of version schemes supported by my proposal.

Perhaps it's better if, when we do XML, we use XSD to describe the version schemes.

We define a set of simple/complex types that people have to extend, and the engine can then convert it to a parser/verifier/order implementation/representation using the base classes.

The parser would use the 'prefix' values and the type attributes to determine what kind of token is next, in the version string. It would then build an XML DOM that can be validated against the XSD, which can have extra rules. Anyway, UDDI tried to do something similar for random applications - to have the API specs in a uniform format so you could generate an application around reusable components, but that didn't work out and AFAIK the specs are

just human readable documents. If somebody wants to create a parser/validator/sorter for a version spec, there should just be enough documentation for them to do it.

References and Related Material

[0] [OSGi Service Platform Release 4 Version 4.1 Core Specification](#), §3.2.4 "Version" and §3.2.5 "Version Ranges" on page 38, §3.5.3 "Bundle-Version" on page 46, §6.1.26.5 "Version.compareTo()" on page 200