

# Mercury Version Ranges

On this page I am trying to capture different designs, problems and solutions around Version Range processing in Mercury project. Please also read the dev@ list discussion on the subject: <http://www.mail-archive.com/search?q=mercury+version+range&l=dev%40maven.apache.org>

---

## Design

Mercury uses a standard OSGi definition of a version range, defined in OSGi core specs 4.1, page 39 in April-2007 PDF available on OSGi site - <http://osgi.org>.

The gist of it is:

- [1.2.3, 4.5.6) 1.2.3 <= x < 4.5.6
- [1.2.3, 4.5.6] 1.2.3 <= x <= 4.5.6
- (1.2.3, 4.5.6) 1.2.3 < x < 4.5.6
- (1.2.3, 4.5.6] 1.2.3 < x <= 4.5.6
- 1.2.3 1.2.3 <= x

 Please note, that multi-range definition, like `<version>[1,3),(3,8]</version>` is gone.

## Dependency Builder flow

**Dependency Builder** component of Mercury is responsible for

- given initial GAV, building the "dirty" dependency tree
- resolving conflicts in the tree and producing a classpath for a particular scope

It uses **RepositoryReader** components to get all the data from repositories, in particular call **readDependencies()** produces a List<ArtifactBasicMetadata> which is a POM-ordered list of **GAV\*s**. Each **\*V** then is treated as a version range and a call is issued **readVersions()** for each range to produce all found range GAV's

## Version Range

Main discussion point is Maven notion of **release** vs. **snapshot**. We are used to lineups such as:

### Maven version lineup

**1.2, 1.3-SNAPSHOT, 1.3-alpha-1, 1.3-alpha-2, 1.3-beta-1, 1.3**

Question to answer: what constitutes the **range [1.2,1.3)** and how flexible this implementation should be?

Various options:

1. 1.2, 1.3-SNAPSHOT, 1.3-alpha-1, 1.3-alpha-2, 1.3-beta-1
2. 1.2, 1.3-alpha-1, 1.3-alpha-2, 1.3-beta-1
3. 1.2

All 3 options have use cases behind them, how general those use cases are and should all 3 options be implemented - remains open.

## Solution to the above problem - Quality Range

In order to cover all the use cases regarding the upper limit of the version range, I introduced a **Quality Range** to Mercury. This allows us to configure **Version Range** to accept

1. **QualityRange.ALL**
2. is split into two
  - a. **QualityRange.ALPHA**
  - b. **QualityRange.BETA**
3. **QualityRange.RELEASE**

As a result all 3 cases could be configured into Mercury

Everybody seem to agree that **plugin versioning** should be identical to regular dependency versioning, so the discussion really applies to both.

**Also on this subject:**

**Mark Hobson wrote:**

Welcome to the debate 😊 I say that a version range should only resolve to a snapshot if it is included as an explicit boundary. See <http://jira.codehaus.org/browse/MNG-G-3092>

You may also be interested in the other issues on my hit list for being able to use version ranges in Maven: <http://jira.codehaus.org/browse/MNG-2994> <http://jira.codehaus.org/browse/MRELEASE-262> <http://jira.codehaus.org/browse/MRELEASE-318>

---

## Alternative options

**Michael McCallum wrote:**

*To be well rounded we should consider other approaches to dependencies*

*its worth having a look at how gentoo does versioning with ranges and slots... <http://www.gentoo.org/h> <http://devmanual.gentoo.org/general-concepts/dependencies/index.html> <http://devmanual.gentoo.org/general-concepts/slotting/index.html>*

---

## Use Cases

**Stephen Connolly wrote:**

*A slightly unrelated question:*

*Will there be support for version ranges with many parts (not just the three parts that maven currently has) so that*

*[1.0.0.0.22,] will not pick up 1.0.0.0.9*

✓ The following unit tests pass fine in Mercury:

```
public void test6Digits()
    throws VersionException
    {
        String rangesS = "[ 1.0.0.1.2.1 , )";
        range = new VersionRange( rangesS );

        assertTrue( "1.0.0.1.2.1 did not match the range "+rangesS,
            range.includes( "1.0.0.1.2.1" ) );
        assertTrue( "1.0.0.1.2.2 did not match the range "+rangesS,
            range.includes( "1.0.0.1.2.2" ) );
        assertTrue( "1.0.0.1.3.0 did not match the range "+rangesS,
            range.includes( "1.0.0.1.3.0" ) );
        assertFalse( "1.0.0.1.2.0 does matches the range "+rangesS,
            range.includes( "1.0.0.1.2.0" ) );
        assertFalse( "1.0.0.1.2.1-alpha-1 does match the range
            "+rangesS, range.includes( "1.0.0.1.2.1-alpha-1" ) );
        assertTrue( "1.0.0.1.2.2-alpha-1 does not match the range
            "+rangesS, range.includes( "1.0.0.1.2.2-alpha-1" ) );
    }

public void testAlphaNumeric()
    throws VersionException
    {
        String rangesS = "[1.0.0.0.22,)";
        range = new VersionRange( rangesS );

        assertFalse( "1.0.0.0.9 does match the range "+rangesS,
            range.includes( "1.0.0.0.9" ) );
    }
}
```

**Michael McCallum wrote:**

3). Declaration [2.0, 2.1) should exclude 2.1-SNAPSHOT, but include 2.1-alpha-1, etc

*Should most definitely not include 2.1-alpha-1 consider this scenario...*

*module Z released as 2.X*

*a dependent module Y specifies X [2,3)*

*you now make a breaking change and release the alpha version of Z 3.0-alpha-1*

*and BAM module Y is using it when it explicitly said I only want major version 2*

**Michael McCallum wrote:**

2). I strongly feel that failing any explicit ranges, containing snapshots is a good thing. For instance, dependency declaration 1.2-SNAPSHOT is a range by definition, so I'd rather fail anything like [1.2-SNAPSHOT,2.0) or [1.0,1.2-SNAPSHOT)

*if you don't allow 1.2-SNAPSHOT how do you actually include them*

*lets assume that 1-SNAPSHOT < 1-alpha < 1-beta < 1 < 1.1 < 1.1.1*

and i say [1,2) then -SNAPSHOT, alpha and beta will not match

I always start my versions at 1.1, 2.1, 3.1 for the lower bound... otherwise you end up with not being able to use the first snapshot of a new major version in a range

And I use the -! syntax for the upper bound which stops the next major versions first snapshot from creeping into a range for the previous major version

**Mark Hobson wrote:**

*Regarding Michael's suggestion of using repositories to control whether ranges resolve to snapshots or not: the problem with this is that it's an all or nothing approach.*

*For example, say I need to fix a bug in Project A 3.0 that depends on Project B 2.0, amongst many other dependencies. I take A 3.0 and determine that the bug is in B 2.0, so I want to update the dependency of B in A to 2.1-SNAPSHOT. Assuming that this range was initially declared as B[2.0,3.0), using the repository approach I would just enable snapshot repositories for the range to resolve to my new work-in-progress B 2.1-SNAPSHOT. This works, but it would also open the gates for all other ranged dependencies to resolve to snapshots too, which I certainly don't want. Whereas if ranges behaved as I've described, then we would just update B to [2.0,2.1-SNAPSHOT) during development and then reinstate [2.0,3.0) once the fixed B 2.1 has been released.*