

MAVEN PROJECT BUILDER SPECIFICATION

SHANE ISBELL

CONTENTS

1. Model Transformations	1
1.1. Canonical Data Format	1
2. General Inheritance Rules	2
2.1. Constructing	2
2.2. Sorting	3
2.3. Model Containers	3
2.4. Mixins and Multiple Inheritance	4
3. Maven Project Inheritance Rules	5
3.1. Inheriting Version and Group Ids	5
3.2. Properties Excluded From Being Overridden	5
3.3. Properties Excluded From Inheritance	5
3.4. Marking Containers as Final (Or Not Inherited)	6
3.5. Artifact Inheritance (Model Container)	6
3.6. Id Inheritance (Model Container)	7
3.7. Plugin Configuration Inheritance	8
4. Management Rules	8
4.1. Dependency/Plugin Management	8
5. Interpolation Rules	9
5.1. Type of Properties	9
5.2. Processing Rules	10
5.3. Interpolation and Profiles	10

1. MODEL TRANSFORMATIONS

1.1. **Canonical Data Format.** Maven supports a canonical data format for the pom that includes 465 model properties (we refer to each ordered pair of uri/values as a model property).

`http://apache.org/maven/project/modelVersion`

`http://apache.org/maven/project/groupId`

`http://apache.org/maven/project/artifactId`

So a valid set would contain ordered pairs:

$\mathcal{A} = \{\langle "http://apache.org/maven/project/groupId", "org.apache.maven" \rangle, \langle "http://apache.org/maven/project/artifactId", "mavencore" \rangle \dots\}$.

Technically \mathcal{A} is also ordered.

Anyone is free to create a transformer from any another format (yaml, .NET projects files, etc) to this canonical data format, giving them all the benefits of project inheritance and interpolation that Maven uses in building a project.

1.1.1. *Collections.* A model property may be specified as a collection, which allows specialized join rules for adding model properties. Collections of collections are allowed.

`http://apache.org/maven/project/build/plugins#collection`

`http://apache.org/maven/project/build/plugins#collection/plugin/executions#collection`

`http://apache.org/maven/project/profiles#collection`

There are 31 collections within the canonical data format.

1.1.2. *Sets.* A model property may be specified as a set, which means that model properties are not duplicated. Generally sets are only used on configuration properties of the plugins.

`http://...pluginManagement/plugins#collection/plugin/configuration#set`

`http://...plugins#collection/plugin/configuration#set`

1.1.3. *Singletons.* Any model property not defined as a collection or set is a singleton, This means that only one entry containing the model property's URI is allowed in the transformed list.

2. GENERAL INHERITANCE RULES

General inheritance rules are those rules applied to a list of model properties, independent of the domain context. The framework delegates domain specific inheritance rules to ModelTransformers provided to it by the invoking application. These will be covered in the next section, under *Maven Project Inheritance Rules*.

2.1. **Constructing.** Basic construction rules are as follows

- (1) Let there be a collection of domain models (poms) denoted by set \mathcal{D}_i , where for some $n \in \mathbb{N}$ the collection is ordered from most specific to least specific $\mathcal{C} = \{\mathcal{D}_0, \mathcal{D}_1, \dots, \mathcal{D}_n\}$. \mathcal{D}_n is the SuperPom and must be contained within \mathcal{C} , even if it is the only element.
- (2) Let p_j be an ordered pair (or model property). In the case of the pom, $j = nodes + properties$ of the pom. Define t as a function operating on elements of \mathcal{C} that transforms each element to a set of model properties. $\mathcal{D}'_i = t(\mathcal{D}_i) = \{p_0, p_1, \dots, p_m\}$. We end up with a transformed collection of domain models: $\mathcal{C}' = \{\mathcal{D}'_0, \mathcal{D}'_1, \dots, \mathcal{D}'_n\}$.

- (3) Next domain specific rules are applied (See section 3). Let tr be a domain transform rule: $\forall_j \forall_i \mathcal{A}_{i,j} \subseteq \mathcal{D}'_i$ such that $\mathcal{A}'_{i,j} = \{tr(p_0), tr(p_1), \dots, tr(p_n)\}$. tr may change either the model property URI or value or may transform the property to a null value (remove it) or it could add additional model properties not from the original set. We are left with $\mathcal{C}'' = \forall_j \forall_i \bigcup_{i,j} (\mathcal{D}'_i \cup \mathcal{A}'_{i,j} - (\mathcal{D}'_i \cap \mathcal{A}_{i,j}))$. Thus \mathcal{C}'' is just a set of transformed and untransformed model properties from all of the domain models \mathcal{D}_i . These model properties are still ordered from most specialized model to least.
- (4) Model properties are now sorted (see section 2.2). Collections and sets are left in reverse order.
- (5) Model container rules are applied for collections and sets. The general sorting in the previous step doesn't know how to handle collections and sets and needs to delegate back to the domain specific model containers (Sections 3.4 and 3.5)
- (6) Model properties are sorted (see section 2.2) again. This is to maintain the original order of collections.
- (7) Interpolates the model properties (Section 5)
- (8) Applies dependency management rules
- (9) Applies plugin management rules

2.2. Sorting. Let \mathcal{C}'' be the original set of model properties and let \mathcal{C}''' be the new set, which is initially empty. Now iterate over each element (model property) of \mathcal{C}'' placing the model property into set \mathcal{C}''' according to the following rules:

- (1) Let u_i be the uri from model property $\langle uri, value \rangle_i$. If $u_i = \text{baseUri}$, then it is placed first in the list. In the case of Maven, <http://apache.org/maven/project> is the baseUri and defines the top node name.
- (2) If u_i is not within any of the model properties contained within \mathcal{C}''' then place the model property into \mathcal{C}''' . This rule only allows one singleton into the set: <http://.../project/groupId> and since \mathcal{C}'' is sorted in order of most specialized to least specialized, only the most specialized pom values will be maintained.
- (3) If u_i contains a value of `#collection` or `#set` but does not end with `#collection` or `#set` then then place the model property into \mathcal{C}''' , at the position of its first (and only) parent. For example, <http://.../project/build/plugins#collection> would have been added in the previous step (because it was not contained in \mathcal{C}''') but this step would exclude any additional model properties containing <http://.../project/build/plugins#collection>. However, all model properties containing uri <http://apache.org/maven/project/build/plugins#collection/plugin> would be added just below its collection. Only one node of a collection type is maintained but multiple children within that collection are allowed.

2.3. Model Containers. In addition to the general inheritance rules, there is also the concept of Model Containers, which allow the framework to delegate to

specific model container implementations the decision of whether `#collections` and `#sets` should be joined, deleted, or have no operation applied. This will be covered more fully in section 3.

2.4. Mixins and Multiple Inheritance. Currently, Maven 3.0 supports linearized inheritance, making mixins and multiple inheritance easy. Support for multiple inheritance would require an additional to the pom, within the parents section.

```
<parents>
  <parent>
    <groupId>org.apache.maven.shared</groupId>
    <artifactId>maven-shared-components</artifactId>
    <version>9</version>
  </parent>
  <parent>
    <artifactId>maven</artifactId>
    <groupId>org.apache.maven</groupId>
    <version>3.0-SNAPSHOT</version>
  </parent>
</parents>
```

In the case above, the child pom's model properties would be first in the set, followed by the model properties of *maven-shared-components*; then *maven* project's model properties and finally by the SuperPom's model properties. So from the framework's perspective there is little difference between multiple inheritance and single inheritance.

Mixins would function the same as multiple/single inheritance:

```
<mixins>
  <mixin>
    <groupId>org.apache.maven</groupId>
    <artifactId>dependency-mixin</artifactId>
    <version>1</version>
  </mixin>
  <mixin>
    <groupId>org.apache.maven</groupId>
    <artifactId>repository-mixin</artifactId>
    <version>2</version>
  </mixin>
</mixins>
```

The only difference between a parent project and a mixin is that the mixin is abstract (not a complete model).

3. MAVEN PROJECT INHERITANCE RULES

These rules outlined in this section are provided in the `PomTransformer` class. The `maven-shared-model` framework will delegate to this transformer for the processing of the Maven specific domain model rules.

3.1. Inheriting Version and Group Ids. If `project.version` is not specified within the child pom, the child pom will use the `project.parent.version` as its own version. Similarly, if `project.groupId` is not within the child pom, the child pom will use the `project.parent.groupId` as its own `project.groupId`.

3.2. Inheriting URLs.

3.3. Properties Excluded From Being Overridden. If the child project defines any of the properties below, they are not overridden by or joined with elements of the parent pom(s).

- (1) `project.build.resources`
- (2) `project.build.testResources`
- (3) `project.pluginRepositories`
- (4) `project.organization`
- (5) `project.licenses`
- (6) `project.developers`
- (7) `project.contributors`
- (8) `project.mailingLists`
- (9) `project.ciManagement`
- (10) `project.issueManagement`
- (11) `project.distributionsManagement.repository`
- (12) `project.distributionsManagement.snapshotRepository`
- (13) `project.distributionsManagement.site`

3.4. Properties Excluded From Inheritance. A child project does not inherit the following properties from its specified parent project¹. All other properties are inherited, unless otherwise noted below.

- (1) `project.parent`
- (2) `project.name`
- (3) `project.packaging`
- (4) `project.profiles`
- (5) `project.version`
- (6) `project.groupId`
- (7) `project.prerequisites`
- (8) `project.distributionManagement.relocation`

¹Technically, `project.version`, `project.groupId` and `project.artifactId` are not inherited from the parent pom. They do, however, have the values of `project.parent.version`, `project.parent.groupId` and `project.parent.artifactId` implicitly applied from the same pom.

3.5. Marking Containers as Final (Or Not Inherited). A parent project can set an inherited property within the following elements of the pom. This will mark the container as final, thus preventing inheritance:

- (1) project.build.plugins.plugin
- (2) project.build.plugins.plugin.executions.execution
- (3) project.build.pluginManagement.plugins.plugin
- (4) project.build.pluginManagement.plugins.plugin.executions.execution
- (5) project.profiles.profile.build.plugins.plugin
- (6) project.profiles.profile.build.plugins.plugin.executions.execution
- (7) project.profiles.profile.build.pluginManagement.plugins.plugin
- (8) project.profiles.profile.build.pluginManagement.plugins.plugin.executions.execution
- (9) project.reporting.plugins.plugin
- (10) project.reporting.plugins.plugin.reportSets.reportSet
- (11) project.profiles.profile.reporting.plugins.plugin
- (12) project.profiles.profile.reporting.plugins.plugin.reportSets.reportSet

Some examples demonstrating use within the project model:

```

<plugin>
  <groupId>org.apache.maven</groupId>
  <artifactId>sample</artifactId>
  <version>1.0</version>
  <inherited>>false</inherited>
</plugin>
<plugin>
  <groupId>org.apache.maven</groupId>
  <artifactId>sample</artifactId>
  <version>1.0</version>
  <executions>
    <execution>
      <inherited>>false</inherited>
    </execution>
  </executions>
</plugin>

```

3.6. Artifact Inheritance (Model Container).

3.6.1. *Defined Nodes.* Within the project there are a number of nodes which contain artifactId, groupId and version. These nodes may be inherited or joined.

- (1) project.dependencies.dependency
- (2) project.build.plugins.plugin
- (3) project.build.plugins.plugin.dependencies.dependency
- (4) project.build.plugins.plugin.dependencies.dependency.exclusions.exclusion
- (5) project.dependencyManagement.dependencies.dependency

- (6) project.build.pluginManagement.plugins.plugin
- (7) project.build.pluginManagement.plugins.plugin.dependencies.dependency
- (8) project.reporting.plugins.plugin
- (9) project.build.extensions.extension

3.6.2. *Rules.* Let the parent project be \mathcal{A} and the child project be \mathcal{B} . Let both $\alpha_i \subset \mathcal{A}$ and $\beta_i \subset \mathcal{B}$ be one of the elements listed above. For example, α_1 would contain all the elements of a project dependency within the parent project.

Both $\alpha_i \subset \mathcal{A}$ and $\beta_i \subset \mathcal{A}$, contain at least the following elements:

- (1) project.groupId (required)
- (2) project.artifactId (required)
- (3) project.version (default value - null)
- (4) project.type (default value - jar)
- (5) project.classifier (default value - null)

(1-3) may be values referencing project.parent.groupId, project.parent.artifactId, project.parent.version, where they are not explicitly defined.

More precisely we have:

$$\forall_i \forall_j \alpha_i = \{ \langle groupId, value_j \rangle_i, \langle artifactId, value_{j+1} \rangle_i, \langle version, value_{j+2} \rangle_i, \dots \}.$$

Now define the following rules:

- (1) $R_1 \equiv groupId(value)^{\alpha_i} = groupId(value)^{\beta_i} \wedge artifactId(value)^{\alpha_i} = artifactId(value)^{\beta_i} \wedge type(value)^{\alpha_i} = type(value)^{\beta_i} \wedge classifier(value)^{\alpha_i} = classifier(value)^{\beta_i}$
- (2) $R_2 \equiv version(value)^{\alpha_i} = version(value)^{\beta_i}$

The inheritance rules are JOIN, NOP, and DELETE:

- (1) $R_1 \wedge R_2 \Rightarrow \mathcal{B}_{new} = \mathcal{B} \cup \alpha_i - (\alpha_i \cap \beta_i)$
- (2) $\neg R_1 \Rightarrow \mathcal{B}_{new} = \mathcal{B} \cup \alpha_i$
- (3) $R_1 \wedge \neg R_2 \Rightarrow \mathcal{B}_{new} = \mathcal{B}$

3.6.3. *Default Group IDs.* To maintain backwards compatibility, the following elements are assigned a default groupId of *org.apache.maven.plugins*, if the groupId is not specified.

- (1) project.build.plugins.plugin
- (2) project.profiles.profile.build.plugins.plugin
- (3) project.build.pluginManagement.plugins.plugin
- (4) project.build.profiles.profile.pluginManagement.plugins.plugin
- (5) project.reporting.plugins.plugin
- (6) project.profiles.profile.reporting.plugins.plugin

3.7. Id Inheritance (Model Container).

3.7.1. *Defined Nodes.* Within the project there are a number of nodes which contain id. Each of the nodes below is an element of a collection, meaning there may be more than one. The ID is used to determine whether the containers should be joined, rather than just added to the collection..

- (1) project.pluginRepositories.pluginRepository
- (2) project.repositories.repository
- (3) project.reporting.plugins.plugin.reportSets.reportSet
- (4) project.profiles.profile
- (5) project.build.plugins.plugin.executions.execution

3.7.2. *Rules.* If an id exists in both the parent and child pom and the ids are equal, then join the nodes, otherwise inherit the node.

3.8. **Plugin Configuration Inheritance.** Plugin nodes are treated as a set. If a child pom contains the same element as a parent pom, then the parent pom element will not be inherited/joined unless the child element contains a property `combine.children="append"`. In this case, it will treat the element as a collection.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <testExcludes combine.children="append">
      <testExclude implementation="java.lang.String">
        **/PersonThreeTest.java
      </testExclude>
    </testExcludes>
  </configuration>
</plugin>
```

If the parent pom contains an element that the child pom does not have, the element will be inherited.

4. MANAGEMENT RULES

4.1. **Dependency/Plugin Management.** Dependency and plugin management are treated the same, so we will only cover dependency management. Our initial set has already been processed for inheritance and interpolated by the time these rules are applied.

Let \mathcal{A} be the set of *project.dependencies.dependency* model containers (model containers are themselves sets of model properties).

Let \mathcal{B} be the set of *project.dependencyManagement.dependencies.dependency* model containers. \mathcal{B} is processed such that each *dependencyManagement* reference within its *uris* is removed. Thus the *uris* exactly match those contained within \mathcal{A} . Call this transformed set \mathcal{B}' .

Now we can apply the same artifact container rules between each \mathcal{B}'_i and \mathcal{A}_j , as those defined in section 3.4.

5. INTERPOLATION RULES

5.1. **Type of Properties.** There are four types of properties in the following order of precedence: maven properties, system (user) properties, project properties, environment (execution) variables.

5.1.1. *Maven Properties.* There are two maven specific properties that can be used: `${basedir}` (or `${pom.basedir}` or `${project.basedir}`) and `${build.timestamp}`. *basedir* denotes the project directory of the executing pom, while *build.timestamp* denotes the time that the build started.

```
<build>
  <directory>${project.basedir}/target</directory>
  <sourceDirectory>${project.basedir}/src/main/java</sourceDirectory>
</build>
```

5.1.2. *System Properties.* These properties are defined on the command line through `-D` option. For instance, `-DjunitVersion=3.8`. These property values take precedence over project and environment properties and will override them.

5.1.3. *Project Properties.* These properties are derived directly from the pom itself: `${project.version}`, `${project.artifactId}`... So in the code snippet below, *project.build.finalname* will be resolved to *maven-3.0-SNAPSHOT*.

Note *pom* is an alias for *project*, so you can also reference the properties through `${pom.version}`, `${pom.artifactId}`... , although *project* is preferred.

These types of properties also include special rules for the *project.properties* section of the pom. The elements under the properties section can directly be referenced, by name, from within other elements of the pom. For example, the *project.properties* section defines *junitVersion*, allowing the *project.build.dependencies.dependency.version* to reference the value by inserting `${junitVersion}`

```
<project>
  <groupId>org.apache.maven</groupId>
  <artifactId>maven</artifactId>
  <version>3.0-SNAPSHOT</version>
  <build>
    <finalName>${project.artifactId}-${project.version}</finalName>
  </build>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junitVersion}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

```

<properties>
  <junitVersion>3.8.1</junitVersion>
</properties>
</project>

```

Keep in mind that if you set `-DjunitVersion=3.8` on the command line, then this value would be used for interpolation, not the pom specified one.

5.1.4. *Environment Properties.* The properties are taken from the environment and hold the lowest level of precedence.

5.2. **Processing Rules.** The pom XML is flattened to a list of model properties (this is part of the inheritance processing). The interpolator property list will be referred to as interpolator properties.

5.2.1. *Preprocessing.* The initial interpolator property list is constructed and sorted in order of maven properties, system properties, project properties and environment properties. Being a list, it contains duplicate property keys that may reference different values. A common example occurs when overriding a pom property through the command line `-D`. So all lower duplicate key values are eliminated, resulting in a set of interpolator properties, where order does not matter.

The maven property `project.basedir` is only added to the initial list if the pom being interpolated is within the build (not a dependency within the repository).

5.2.2. *Pass 1 -withhold using build directories as interpolator properties.* In this pass, the list is preprocessed into a set, but excludes any of the build directories from the interpolator list. In other words, the build directories can be interpolated but they can't be used to interpolate other properties. Interpolating is simply the iteration of all interpolator properties over model properties.

5.2.3. *Pass 2 - set absolute paths on build directories.* At this point, the build directories are completely interpolated but they may or may not contain absolute paths. So each build model property is checked and if it contains a relative path, the absolute path is set based on the `project.basedir` location.

5.2.4. *Pass 3- use build directories as interpolator properties.* In this pass, all model properties that contain a build directory reference are interpolated with the build directory interpolator properties, which were withheld from pass 1. Now all directory paths within the pom contain absolute references.

5.3. **Interpolation and Profiles.** Active profiles are applied prior to interpolation so that any `project.profiles.profile.properties` defined within an active profile can be used as an interpolation property [Still to be implemented]