Status

This RFC is currently in the DRAFT state. Nothing in this RFC has been agreed or confirmed.

Contents

- Status
- Contents
- Introduction
  - Background
- Classification of change requests
  - New content to include in the POM
  - Supports / provides style concepts
  - Versioning related issues
  - Lifecycle related changes
  - Scope related changes
  - Profile activation
  - POM format
  - Mix-ins
- Existing model
- Changes
  - Dual usage
  - XML vs custom DSL
  - Elements vs Attributes
  - Customizing build behavior / One-off projects
  - Custom scopes
  - Build vs Reporting
- Project Object Model
  - <project> element
  - <parent> element
  - <mixin> element
  - <extensions> element
  - <extension> element
  - TODO resolve the inheritance problem
- Appendix 1 - Issues flagged for consideration post-modelVersion 4.0.0

Introduction

The next generation Project Object Model to be used by Maven 5.0+

Background

Maven uses the Project Object Model as a descriptor for the declarative build requirements of a project.

- Maven 1.x used a model which contained a `<modelVersion>3.0.0</modelVersion>` element as an immediate child of the root.
- Maven 2.x / 3.x has used a `<modelVersion>4.0.0</modelVersion>` element.

Due to the way Maven has been implemented, the current release versions will consider any modelVersion other than the one that they target as invalid and will fail to parse the model.
For build time concerns, this is not that major a concern, and in fact may be desirable behaviour, e.g. I should not be able to build a Maven 2.x / 3.x project with Maven 1.x.

Where the modelVersion becomes a constraint, however, is when it comes to transitive dependency resolution. The Maven Central repository has grown in popularity, and now the consumers of the information in central are no longer only Apache Maven. There are other build tools that parse the POM to extract dependency information, e.g. Apache Buildr, Gradle, Apache Ivy, sbt, etc. As these build tools are not under the control of the Apache Maven project, we risk breaking their ability to parse the POM as a unit of dependency expression if we modify the pom schema or model version.

While we could change the schema if we “forked” the central repository, the experience from the previous repository fork (for the Maven 1.x / Model Version 3.0.0 to Maven 2.x / Model Version 4.0.0 transition) was traumatic and a repeat is generally considered to be a Bad Plan™.

The result of all this is that the Apache Maven project has been unable to evolve our POM to reflect the new needs.

The current plan for a Path Forward™ uses three legs:

1. We keep deploying modelVersion 4.0.0 poms to the repository as a best effort expression of the dependency information of artifacts such that legacy clients can continue to consume artifacts deployed with non-legacy clients.
2. We deploy a dependency-only model using a defined contract for forwards compatibility (to allow for future evolution) using a different file extension (see Project Dependency Trees schema)
3. The POM then becomes a build-time only concern and does not need to be deployed to the repository - except for those cases where the pom may be used as either a parent or a mix-in

This page will represent (TODO replace “will represent” with “represents” when near finalised) the specification for the next modelVersion of the POM to be used by Maven.

Classification of change requests

There are currently 0 issues flagged as either waiting for a major version bump in Maven because they are a behavioural change or waiting for a modelVersion bump because they change the POM schema. This section aims to classify and summarise the changes requested by users / developers in order to better understand the rationale for the proposed new POM schema.

New content to include in the POM

There are six general sub-themes around content to include in the POM.

The following issues look to add content for documentational purposes. This content would be consumed both by developers reading the POM “by hand” as well as by more automated tooling such as the Maven Site generation

- **MNG-50** - Getting issue details... STATUS looks for the ability to include links to the coding standards and formatting rules that a project uses.
- **MNG-3726** - Getting issue details... STATUS looks for the ability to document IRC channels. More generally, if IRC channels are documented, other kinds of instant messaging and social media channels should be documented.
- **MNG-4921** - Getting issue details... STATUS looks for the ability to document quality management services (such as sonar) similar to how we allow defining continuous integration services (such as Jenkins)

The following issues look to add content to assist using maven on a specific project

- **MNG-2916** - Getting issue details... STATUS looks for the ability to define information to display to the user if they invoke maven with no goals specified and no default goal defined in the POM. Additionally the message should be customizable per profile
- **MNG-5563** - Getting issue details... STATUS looks for a way to validate plugin configuration. An interesting thought experiment would be to allow a POM to be parameterized with some parameters requiring input at invocation time such that Maven would always ask for that parameter (using a custom prompt) if it wasn’t supplied.

The following issues concern configuration that needs to be shared between plugins

- **MNG-4506** - Getting issue details... STATUS (probably could be handled as maven-site-plugin configuration) looks to allow defining a different site deployment URL for SNAPSHOT versions of the project compared with release versions)
- **MNG-2216** - Getting issue details... STATUS looks for the ability to define the default encodings to be used when reading files (and optionally when writing files)
- **MNG-3608** - Getting issue details... STATUS looks for the ability to define the default encodings to be used when writing the site reporting files
The following issues concern providing explanations of dependencies within the POM

- **MNG-3879** - Getting issue details... STATUS looks for the ability to provide comments within the `<dependency>` tags as consumers are often unclear of the rationale for inclusion of some dependencies
- **MNG-5926** - Getting issue details... STATUS looks for the ability to provide comments within the `<dependency>` and `<excludes>` tags for the same reason as MNG-3879

The following issue concerns property evaluation

- **MNG-5900** - Getting issue details... STATUS looks for the ability to declare property references that would be evaluated before pulling in parent / mix-in / etc such that those property references could be used to control the parent / mix-in being pulled in.

The following issues repeat / revert changes that previous experience has deemed to be a mistake. As such the current opinion is that these issues should not be fixed.

- **MNG-5657** - Getting issue details... STATUS looks to move the `<distributionManagement>` section out of the pom and into settings.xml. It is unclear how this would work as different projects would need different distribution management details. The only use case where this becomes valid is when deploying a custom fork of a project to an internal repository...
- **MNG-5659** - Getting issue details... STATUS looks to resurrect profiles.xml which was generally considered to be a mistake.

**Supports / provides style concepts**

The following issues are all essentially the same theme, namely look to add additional classes of dependency information to the dependency graph.

- **MNG-177** - Getting issue details... STATUS looks to provide a mechanism for a dependency to declare itself as being a drop-in replacement for another artifact.
- **MNG-5652** - Getting issue details... STATUS is a superset of MNG-177 and basically defines a new type of dependency graph declaration which indicates that an artifact is a drop-in replacement for another artifact.
- **MNG-1977** - Getting issue details... STATUS looks to provide a mechanism to globally ban specific dependencies. The driving use case for this is that dependency A and dependency B are equivalent and the duplicate content needs to be resolved by removing one from the dependency graph.
- **MNG-2316** - Getting issue details... STATUS looks to provide a mechanism for a dependency to declare itself as being a drop-in replacement for another artifact.
- **MNG-5867** - Getting issue details... STATUS looks to provide a mechanism for a dependency to declare itself as being a drop-in replacement for another artifact.

**Versioning related issues**

There is no specific set of themes here:

- **MNG-624** - Getting issue details... STATUS looks for the ability to have automatic parent versioning. Some attempts in core have been made to enable the project version to be deterministic from e.g. source control such that the pom does not need to be modified in order to release.
- **MNG-4173** - Getting issue details... STATUS looks to force users to always specify the versions of plugins.
- **MNG-5517** - Getting issue details... STATUS looks to change the version range syntax from the mathematical range syntax used by Maven.

**Lifecycle related changes**

Two main themes around lifecycle changes

The following issues relate to trying to solve dependency issues within a multi-module reactor where one module consumes artifacts at one point in the lifecycle which are produced at a different point in the lifecycle by a different module.

- **MNG-193** - Getting issue details... STATUS looks for a way to declare what the outputs of a plugin will be
wants to find a way around the "jar == target/classes prior to package phase"

The following issues relate to specification of the lifecycle itself:

- **MNG-5384** - Getting issue details...  STATUS wants to work on simple multi-module projects

  looks to add a layer of indirection to the lifecycle bindings such that, say the
  "compile" phase could be bound to a generic "compiler" goal and then another layer could define that for a specific project / packaging
  the "compiler" generic goal would be fulfilled by a specific plugin's goal execution.

- **MNG-3522** - Getting issue details...  STATUS looks to define a specific plugin execution order within a phase. The driver for
  this use case is that the lifecycle cannot be customised so when a module needs a complex lifecycle in order to ensure correct inter-
  plugin execution order

  looks to introduce more advanced constraints on the lifecycle, such as "finally" concepts as well as fork-points in the lifecycle, such as "either install or deploy but not both"

**Scope related changes**

There is no specific set of themes here:

- **MNG-1867** - Getting issue details...  STATUS looks to remove system scope.

  looks to introduce a scope that would allow mix-ins for dependency Management

**Profile activation**

The following issues are all focused on gaps in profile activation:

  looks to define profile deactivators which would be the inverse of profile activators. Some of the use cases seem a bit hacky, but
  as a principal being able to express activation via an inverse condition can be simpler for users to comprehend.

- **MNG-3326** - Getting issue details...  STATUS looks to be able to have profiles activated based on the project version

  looks to make the activators into an extension point.

**POM format**

The following issues look to address deficiencies (perceived or otherwise) in the modelVersion 4.0.0 POM format:

- **MNG-3397** - Getting issue details...  STATUS looks to switch to attributes for some of the more annoying verbosity in the POM

  looks to switch to attributes for some of the more annoying verbosity in the POM

- **MNG-5653** - Getting issue details...  STATUS looks to switch from XML to a custom DSL

  looks to move the build/pluginManagement tag to the root level.

**Mix-ins**

The following issues look for mix-ins that allow content for the POM to be included from other sources:

- **MNG-5102** - Getting issue details...  STATUS looks for general purpose mix-ins

  looks for an explicit pluginManagement scoped mix-in

**Existing model**

The existing 4.0.0 model POM has the following high-level structure:
The major critiques of the existing model are:

- Overly verbose and repetitive - the main pain point is that the groupId, artifactId, etc are not specified as attributes
- "I hate XML" - our current thinking is that this is really just a catch-all complaint from people who:
  - Don't like the schema / feel the schema is overly verbose
  - Want to produce an imperative build from a declarative build tool
  - Are toiling for fun and profit
- Poorly specified dependency graph resolution
- "Magic" inheritance - it can be difficult to determine how inheritance will affect the build.
The other issue with the existing model is that it is being used for two distinct purposes and as such finds it difficult to be a master of both:

- The 4.0.0 POM serves as a declarative description of the build process for a project
- The 4.0.0 POM serves as a description of the project dependency graph.

The vision of the 4.0.0 POM was that all projects would be cut from a series of standard templates (a.k.a. packaging):

- Each template would define the appropriate lifecycles and phases of those lifecycles (hopefully most templates/packagings would be sufficiently served by the three default lifecycles: default, clean and site) and each template/packaging would define the plugin bindings against the lifecycle phases.
- Where a project needed a customized build process, the build engineer would initially explore how to develop the build process by customizing the bindings of an existing template/packaging.
- Once the build engineer had determined the correct generic process for building this type of project, the build engineer would then solidify this build process into a custom template/packaging.

In this vision, almost all 4.0.0 POMs should basically consist of the following structure:

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent> <!-- most projects should inherit from a parent pom of some sort -->
    <groupId>...</groupId>
    <artifactId>...</artifactId>
    <version>...</version>
    <relativePath/>
  </parent>
  <groupId>...</groupId>
  <artifactId>...</artifactId> <!-- most projects should inherit the parent's groupId -->
  <version>...</version>
  <packaging>!!-- THIS IS THE IMPORTANT BIT--></packaging>
  <dependencies> ...
    <dependencies> !!-- Here is the project dependencies -->
    </dependencies>
  <build> !!-- no custom plugin configuration or bindings -->
    <extensions> !!-- this is only needed if not inherited from the parent -->
      <extension>
        <groupId>...</groupId>
        <artifactId>...</artifactId>
        <version>...</version>
      </extension>
    </extensions>
  </build>
</project>
```

In other words, when using the 4.0.0 POM in accordance with its initial vision, there should be at most 25 lines of boilerplate above the specification of the project dependencies and in the ideal case that boilerplate can be reduced to ~14 lines which specify:

- the versions of Maven which the POM is compatible with (the modelVersion)
- the parent to inherit from (3 lines of information due to the use of XML elements instead of attributes)
- the identity of this project (2 lines of information if inheriting the groupId from the parent)
- the template/packaging that this project is built with

When we inspect real world POMs however, we see that this pattern is almost never followed. Instead of producing custom templates/packaging most projects instead just fight with a standard template/packaging. The end result of this kind of fighting is POMs that run into the 10,000+ LOC levels with many plugin bindings and overloading of an existing lifecycle binding and profiles used to enable additional side-build processes. The reasons cited for these long POMs include:

- "It is too hard to make a custom template/packaging"
- "This is a one-off project, we will never make another of this type, therefore it doesn't make sense to produce a custom template/packaging"
Changes

This section details the rationale for all the changes to the POM format.

Dual usage

The most important change for the 5.0.0 POM is to split the dual usage:

- The 5.0.0 POM will be used as a declarative description of the build processes of the project.
- The description of the project artifact dependency graphs will be provided by the Project Dependency Trees schema proposal.

DECISION: The POM is for Building, the Project Dependency Trees is for consumption of artifacts

AFFECTS:

XML vs custom DSL

The project dependency trees schema will be XML because that is designed to be a machine generated document that is for consumption primarily by machines but needs to remain easily parsable by humans. The choice of XML is dictated by the requirement to enable multiple tools to have a level of forward compatibility and, at this time, the only cross-technology tool that can deliver a mapping is XSLT. For this reason the Project Dependency Trees schema will be an XML format.

As the 5.0.0 POM will only be used by Maven, and as the 5.0.0 POM will require Maven 5.0+ to build, there is no longer a strict requirement to retain the XML format for the 5.0.0 POM.

There are, however, a number of advantages to continuing with the XML based format at least for the 5.x release train of Maven:

- Most editors already have syntax highlighting and completion support for XML, e.g. when closing an element
- A significant number of editors can use the XML schema to provide enhanced completion support, e.g. providing contextual suggestions for elements and attributes
- Familiarity of the existing user base, e.g. the current users of Maven are already used to the XML based-syntax
- Reduces the number of code paths to allow Maven 5.0+ to parse the 4.0.0 POM, e.g. it will significantly aid adoption of Maven 5.0+ if you can build Maven 2/3 projects with Maven 5.0+

The single biggest reason for retaining XML, however, is that we expect the build model will need to evolve. With the Project Dependency Trees schema, we need to provide for backwards compatibility (i.e. newer clients need to be able to parse older schemas) and limited forward compatibility (i.e. older versions of Maven will not be able to build newer POM modelVersion). Ideally we want the range of backwards compatibility to reach back as far as the 4.0.0 POM. Retaining XML as a POM format allows for technology such as XSLT to be used to map a 5.0.0 POM into a 5.3.0 POM which would thus reduce the number of parsers that would be required to be included within Maven (we will still need a custom parser for the 4.0.0 POM, and it is likely that Maven 6.0+ would need custom two parsers one for the 4.0.0 POM and one for the 5.x.y POMs). A custom DSL would force tool vendors to produce syntax parsers for each and every model version.

DECISION: The 5.0.0 POM will be XML

AFFECTS: MNG-6061 - Getting issue details... STATUS

Elements vs Attributes

There seems to be universal agreement to use attributes where possible. The reason for choosing elements in the 4.0.0 was purely a technical limitation of the Modello toolchain.

DECISION: The 5.0.0 POM will use XML attributes for data that cannot have child data. At a minimum the groupId/artifactId/platformId/version/classifier/type information of project/parent/dependencies/plugins/extensions will be defined using attributes.

AFFECTS: MNG-3397 - Getting issue details... STATUS MNG-5653 - Getting issue details... STATUS

Customizing build behavior / One-off projects

The term packaging in the 4.0.0 POM is used for two distinct purposes: defining the type of the primary artifact and defining the base template of the build process. The Project Dependency Trees schema removes the concept of a primary artifact by providing the dependency trees of all attached artifacts, thus a single Maven project that produces a .jar, .war and even say a secondary "skinny" .war will have the appropriate
dependency trees for each artifact declared in the PDT. This contrasts with the 4.0.0 POM which only defined the dependencies of the primary artifact and relied on build tooling convention to infer what contextual transitive dependencies should be extracted by consumers from the POM.

Thus, in the 5.0.0 POM we only really want to specify the template for the lifecycles and default bindings. Given that the use case for this data is purely as a template, it makes sense to change the name to `template`.

**DECISION:** The 5.0.0 POM will use the term `template` rather than `packaging`.

**AFFECTS:**

One of the requirements that a lot of projects have is cross-cutting inheritance. There is general agreement that mix-ins are the way to achieve this.

**DECISION:** The 5.0.0 POM will allow for mix-ins

**AFFECTS:**

The evidence of the use of Maven shows that there seems to be a significant number of projects that believe themselves to be "one-off" projects that will not benefit from expressing the build logic in a reusable template. We need to provide a way to allow projects to easily change their effective lifecycle and plugin bindings

**DECISION:** The 5.0.0 POM will provide the ability to define custom lifecycles directly within the POM

**AFFECTS:**

**DECISION:** The 5.0.0 POM will provide the ability to override and completely clear the plugin bindings against individual lifecycles

**AFFECTS:**

### Custom scopes

One of the blockers for custom scopes has been the requirement that the 4.0.0 POM be used for both the declarative build description and the consumer's dependency graph construction. Any custom scopes introduced into the POM would either break or confuse clients that relied on the assumed 5 scopes defined in the 4.0.0 POM. The Project Dependency Trees schema removes use case of consumption of the POM by consumers of the artifacts produced by the project. This has the effect of completely removing the limits on scopes. The 4.0.0 scopes will likely remain the conventions as interoperability with older plugins as well as conventions in the default configurations of plugins will simplify their use, but in those cases where a project needs to define and consume its own scopes it should be possible to permit it.

**DECISION:** The 5.0.0 POM will allow the definition and consumption of custom scopes directly within the POM, parent POM, mixins, or templates

**AFFECTS:**

The `system` scope was a Java-centric special scope experiment that hit issues with consumption of dependencies across multiple platforms.

**DECISION:** The 5.0.0 POM will not provide any special case behaviour for a scope named `system`

**AFFECTS:**

### Build vs Reporting

If we look at the two use cases of building with Maven 2/3 there are actually two distinct use-cases:

- Building the project artifacts
- Building the project site/documentation

This was shaped by having two configuration sections in the 4.0.0 POM, `build` and `reporting`. One of the issues with these two sections is that they did not have parity of configurability. Specifically, the `reporting` section did not have a `pluginManagement`. The solution of having the `pluginManagement` from the `build` section apply to the `reporting` section feels incorrect as now the child element of one is affecting another.

**DECISION:** The 5.0.0 POM will treat global plugin configuration defaults as a top level concern and have a tag equivalent to `pluginManagement` at the top level of the POM.

**AFFECTS:**

If we seek to find a generic solution to the split between the `build` and `reporting` sections in the POM, it becomes apparent that these are all really just ways of defining bindings of plugins to the phases of various lifecycles. The `build` section defines the bindings against the default...
lifecycle, while the reporting section defines the bindings against the site lifecycle (Note: this is a simplification as the reporting plugins are actually partly invoked by the site plugin and thus are not actually specifically bound to the lifecycle, rather the site:site goal is bound to the site phase of the site lifecycle and that goal is responsible for invoking the reporting plugin goals)

As a side-effect of making it easier to produce custom lifecycles, we probably need to be able to make it easier to manage the bindings of plugins for the custom lifecycles.

DECISION: The 5.0.0 POM will remove the distinction between build and reporting relying rather on lifecycle specific binding declarations

AFFECTS:

---

**Project Object Model**

**<project> element**

The Project Object Model consists of a top level <project> tag with child elements

```xml
<project modelVersion="5.0.0" [groupId="..."] artifactId="..."
   [version="..."] template="...">
  <parent [groupId="..."] [artifactId="..." [version="..."
               [relativePath="..."/>
  </parent>
  <mixin [groupId="..."] [artifactId="..." [version="..."
               [relativePath="..."/>
  </mixin>
  <extensions [mode="override|inherit">
  ...
  </extensions>
  <lifecycle id="..." mode="override|inherit">
  ...
  </lifecycle>
...
</project>
```

The following are mandatory elements:

- `modelVersion` attribute - containing the model version of the POM.
- `artifactId` attribute - containing the artifactId of the project
- `template` attribute - containing the identifier of the template / packaging that will be used as the initial basis for this project’s conventions

The following are optional elements:

- `groupId` attribute - containing the groupId of the project. If this attribute is missing then the parent element must be present and the groupId will be inherited from the parent project.
- `version` attribute - containing the version of the project. If this attribute is missing then the parent element must be present and the version will be inherited from the parent project.
- `parent` element (cardinality 0-1) - containing at a minimum either the GAV of the parent project or the relative path to the parent project. Where more than the minimum required information is supplied, the additional information will be used to validate the parent project reference.
- `mixin` elements (cardinality 0-N) - containing at a minimum either the GAV of the mix-in project or the relative path to the mix-in project. Where more than the minimum required information is supplied, the additional information will be used to validate the mix-in project reference.
- `extensions` element (cardinality 0-1) - containing the extensions to enable for this project.
- `lifecycle` elements (cardinality 0-N) - containing lifecycle customisations for this project.

**<parent> element**

The parent element identifies the parent project from which conventions will be inherited.
Technically from a schema perspective all attributes are optional, however there are two minimum valid sets of attributes:

- If the `relativePath` attribute is present, no other attributes are required:

  ```xml
  <parent relativePath="...">
  </parent>
  ```

  This indicates that the parent project can be found on disk at the supplied relative path and the conventions should be inherited from that project. Specifying the additional attributes of `groupId` or `artifactId` while the `version` attribute is unspecified will indicate that the build should fail if the project at the supplied relative path does not match the specified `groupId` or `artifactId`. If all three of the `groupId`, `artifactId` and `version` attributes are missing then a mismatch at the supplied relative path will not be fatal as the parent can be resolved from the reactor/repository.

- Specifying the GAV of the parent project:

  ```xml
  <parent groupId="..." artifactId="..." version="..."/>
  ```

  This indicates that the parent project should be resolved from the reactor/repository. If the `relativePath` element is present then in prior to the the reactor/repository the project at the specified relative path will be validated against the supplied `groupId`, `artifactId` and `version` and used in the event of a match.

The following are the attributes:

- `groupId` attribute - containing the `groupId` of the parent project.
- `artifactId` attribute - containing the `artifactId` of the parent project
- `version` attribute - containing the version of the parent project.
- `relativePath` attribute - containing the relative path to the parent project.

**<mixin> element**

The mixin element identifies additional projects from which conventions will be inherited.

```xml
<mixin [groupId="..."] [artifactId="..."] [version="..."]
[relativePath="..."]/>
```

Technically from a schema perspective all attributes are optional, however there are two minimum valid sets of attributes:

- If the `relativePath` attribute is present, no other attributes are required:

  ```xml
  <mixin relativePath="...">
  </mixin>
  ```

  This indicates that the mixin project can be found on disk at the supplied relative path. Specifying the additional attributes of `groupId` or `artifactId` while the `version` attribute is unspecified will indicate that the build should fail if the project at the supplied relative path does not match the specified `groupId` or `artifactId`. If all three of the `groupId`, `artifactId` and `version` attributes are missing then a mismatch at the supplied relative path will not be fatal as the mixin can be resolved from the reactor/repository.

- Specifying the GAV of the mixin project:

  ```xml
  <mixin groupId="..." artifactId="..." version="...">/
  ```
This indicates that the mix-in project should be resolved from the reactor / repository. If the `relativePath` element is present then in prior to the the reactor / repository the project at the specified relative path will be validated against the supplied `groupId`, `artifactId` and `version` and used in the event of a match.

The following are the attributes:

- `groupId` attribute - containing the groupId of the mix-in project.
- `artifactId` attribute - containing the artifactId of the mix-in project
- `version` attribute - containing the version of the mix-in project.
- `relativePath` attribute - containing the relative path to the mix-in project.

### `<extensions>` element

The extensions element identifies extensions to be enabled for this project.

```xml
<extensions [mode="override|inherit"]>
  [extension [groupId="..."] [artifactId="..."] [version="..."]
  [relativePath="..."]/>
  ...
</extensions>
```

There is one attribute:

- `mode` attribute - when specified as `override` then any inherited extensions are ignored and the full set of extensions to be enabled is contained within this element. When specified as `inherit` - the default - then the inherited extensions are merged with the extensions contained within this element. In the case of duplicate `groupId:artifactId` entries, the version declared in this project will take precedence.

There can be 0-N `extension` elements.

### `<extension>` element

The extension element identifies additional projects containing extensions to enable for the project.

```xml
<extension [groupId="..."] [artifactId="..."] [version="..."]
[relativePath="..."]/>
```

Technically from a schema perspective all attributes are optional, however there are two minimum valid sets of attributes:

- If the `relativePath` attribute is present, no other attributes are required:
  ```xml
  <extension relativePath="...">
  </extension>
  ```

This indicates that the extension project can be found on disk at the supplied relative path. Specifying the additional attributes of `groupId` or `artifactId` while the `version` attribute is unspecified will indicate that the build should fail if the project at the supplied relative path does not match the specified `groupId` or `artifactId`. If all three of the `groupId`, `artifactId` and `version` attributes are missing then a mismatch at the supplied relative path will not be fatal as the extension can be resolved from the reactor/repository.

- Specifying the GAV of the extension project:
  ```xml
  <extension groupId="..." artifactId="..." version="...">
  </extension>
  ```

This indicates that the extension project should be resolved from the reactor / repository. If the `relativePath` element is present then in prior to the the reactor / repository the project at the specified relative path will be validated against the supplied `groupId`, `artifactId` and `version` and used in the event of a match.
The following are the attributes:

- groupId attribute - containing the groupId of the extension project.
- artifactId attribute - containing the artifactId of the extension project.
- version attribute - containing the version of the extension project.
- relativePath attribute - containing the relative path to the extension project.

TODO resolve the inheritance problem

We have multiple places where things are coming from...

- parent
- mix-ins
- extensions

we need a model of inheritance that is easy for people to understand.

First stab:

- complete each parent/mixin's project model before inheriting.
  - PROs:
    - We do not have as much complexity, you just help:effective-pom on the parent, and mixins to see what they are defining and hence pulling in
  - CONs
    - What about mixin version conflict? If the parent brings in one mixin and then we explicitly state another version of that mixin and finally explicitly state a 3rd mixin that transitive states a third version of the same mixin.
      If we flatten first, it is very likely that we could have some cruft left over from one of the other versions

Second stab:

- process parent inheritance by injecting pseudo mixin nodes from the parent and then de-dup
  - CONs
    - I'm not even sure what I am saying here

I think we need to build the tree of mixins, resolve the closest version (with tree pruning) and then apply them...

Then after that, we can start to build the tree of extensions...

This is a pain!

TODO write this up... I'm just dumping stuff I have done on the mail thread here to make it easier to collaborate:

```
<project modelVersion="5.0.0" [groupId="..."] artifactId="...
 [version="..."] template="...">
  [<parent groupId="..." artifactId="..." [version="..."]
   [relativePath="..."]/]

  [<mixin groupId="..." artifactId="..." [version="..."]/>]
  [<mixin groupId="..." artifactId="..." [version="..."]/>]
  ...
  [<mixin groupId="..." artifactId="..." [version="..."]/>]

  [<lifecycle id="..." mode="override|inherit">
    <phase id="..." [after="..."] | before="..."]/>
    <phase id="..." [after="..."] | before="..."]/>
```
...<phase id="..." [after="..." | before="..."]/>

</lifecycle>
[<lifecycle id="...">
...
</lifecycle>]
...
[<lifecycle id="...">
...
</lifecycle>]

[<scope id="compile" [mode="override|inherit"]>
  <dependency groupId="..." artifactId="..." [platformId="..."]
    version="..." [classifier="..."] type="..."/><!-- type is mandatory-->
    <dependency groupId="..." artifactId="..." [platformId="..."]
      version="..." [classifier="..."] type="..."/>
...
  <dependency groupId="..." artifactId="..." [platformId="..."]
    version="..." [classifier="..."] type="..."/>
</scope>]
[<scope id="...">
...
</scope>]
...
[<scope id="...">
...
</scope>]

[<extensions [mode="override|inherit"]>
  <extension groupId="..." artifactId="..." version="..."/>
...
</extensions>]
[<plugins [mode="override|inherit"]>
  <!-- this is what pluginManagement was -->
  <plugin groupId="..." artifactId="..." version="...">
...
</plugin>
...
</plugins>]

[<bindings [mode="override|inherit"]>
  <!-- this is what plugins was, we make explicit here that this is the
  binding of executions into the lifecycles -->
  <bindings/>

  [<platform id="..." [mode="override|inherit"]>
    <activation>
      <!-- define how we determine that this platform can be built in the
      current environment -->
      <activation>
        <!-- allow platform specific mixins -->
        [mixin groupId="..." artifactId="..." [version="..."/]
</activation>
<!-- allow platform specific lifecycles -->
[lifecycle id="...">
...
</lifecycle>

<!-- allow platform specific dependencies -->
[<scope>
...
</scope>]

<!-- allow platform specific bindings... but plugin management is from the root only -->
[<bindings>
...
</bindings>]

<!-- allow most of the other root tags except platform and packaging and deployment config -->
</platform>]
[<platform id="...">
...
</platform>]
...
[<platform id="...">
...
</platform>]

<!-- template is only allowed in poms with an id of "parent" or "mixin". It allows a parent/mixin to be used by different template ids and define specialized defaults -->
[<template id="...">
[<mixin groupId="..." artifactId="..." version="...">]
<!-- allow platform specific lifecycles -->
[<lifecycle id="...">
...
</lifecycle>]

<!-- allow platform specific dependencies -->
[<scope>
...
</scope>]

<!-- allow platform specific bindings... but plugin management is from the root only -->
[<bindings>
...
</bindings>]

<!-- allow most of the other root tags except platform and packaging and deployment config -->
</template>]
[<template id="...">
...
Some things that came to mind, in no particular order:

- scope becomes a build time only concern. Thus we can let users define custom scopes in their pom. If we let plugin executions declare scopes to resolve, we no longer need a compiler:testCompile goal as you can just have a second default execution of compiler:compile with different required scopes and different default configuration... bonus win, I can now add many different layers of test-compilation for integration tests, etc... each pulling in different scopes... ditto for surefire/failsafe... yeah integration tests
- we should let the user define lifecycles directly in the Pom (ok, maybe we don't "encourage it")
- mixins can be properly considered... they only affect build time anyway
- Pom doesn't need to be XML any more... (maybe we want to keep XML though... just a less verbose form)
- does Maven 5 build Maven 2/3 projects?

Building the effective build time model would be:

- Start with parent, add in matching packaging from parent, in Pom order, add each mix-in (including matching packaging from mix-in before processing subsequent mix-ins), finally apply local pom.

### Appendix 1 - Issues flagged for consideration post-modelVersion 4.0.0

This is not a perfect query as it includes issues that target behavioural changes outside the scope of modelVersion changes, but all the relevant issues should be a subset of this list

<table>
<thead>
<tr>
<th>Key</th>
<th>Summary</th>
<th>Type</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>🔴</td>
<td>Jira project doesn't exist or you don't have permission to view it.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

View these issues in Jira