Dynamic phases

Status	DRAFT
Version	1
Issue(s)	MNG-5668 - post-<phase> should always be executed after <phase> OPEN
Sources	
Developer(s)	Stephen Connolly

Status

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

Contents

Introduction

Provide a way to control the execution of MOJOs within a phase and guarantee that specific MOJOs will be executed around lifecycle phases

Background

The standard Maven lifecycles have a number of phases with names that start with `pre-` and `post-` as siblings to a main phase, for example:

- `pre-integration-test` and `post-integration-test` are siblings of `integration-test`
- `pre-site` and `post-site` are siblings of `site`
- `pre-clean` and `post-clean` are siblings of `clean`

Most new users assume that as the `pre-` phases will be executed before the main phase (which is correct, but only be the accident of the lifecycle ordering) the `post-` phases will be executed after the main phase much like the finally block in a Java try expression (which is incorrect).

- Most Maven users will invoke the `clean` lifecycle with a command like `mvn clean`, yet to ensure that the lifecycle has run correctly you should really run `mvn post-clean`.
- The `site` lifecycle is slightly better as at least invoking `mvn site-deploy` will afford the `post-site` phase to execute, but this is only for the happy path where the `site` phase completed successfully.
- The `default` lifecycle has the least worst situation because `integration-test` is so long to type that most people will run `mvn verify` which again affords execution only for the happy path where `integration-test` completes successfully. To enable the successful use of the `integration-test` phase therefore requires that MOJOs used in this phase are written in such a way that they never fail and instead provide a second MOJO that can be bound to the verify phase in order to fail the build after the `post-integration-test` phase has completed. This prevents the use of general purpose MOJOs with integration testing and complicates plugin design.

If we look more critically at the lifecycle phases we can also identify a number of phases that are purely present to enable the correct sequencing of MOJO executions:

- `generate-sources` and `process-sources` : it's hard to see why you would ever want to generate the sources and not have them processed, given that any executions bound to `process-sources` is a necessary pre-requisite for compilation
- `generate-resources` and `process-resources`
- `compile` and `process-classes` : it's hard to see why you would want the classes that have not been processed
- `generate-test-sources` and `process-test-sources`
- `generate-test-resources` and `process-test-resources`
- `prepare-package` and `package` : the prepare-package phase was added specifically to enable two phase packaging

The resulting multiude of phases just furthers the complexity for users: they become parallelized by choice.

Proposal

Provide a `pom.xml` only naming scheme for ad-hoc dynamic phases that will enable the `pom.xml` to control execution while restricting this syntax to the `pom.xml` only. The command line would only be able to invoke the explicit lifecycle phases directly.

To clarify, these dynamic phases would only be valid in the `/executions/execution/phase` element of a `<plugin>` in the `pom.xml`

There will be two classes of dynamic phases:

1. Guaranteed execution
2. In phase ordering

Note: please append any alternative syntax proposals to this section

Option 1: `(before:|after:)`*`$phase`*`([`*`$priority`*`])`

This syntax uses two prefixes before: and after: to identify phases that will be guaranteed to run before and after the named phase *`$phase`*. As a result we would be able to deprecate the `pre-integration-test` and `post-integration-test` phases in favour of `before:integration-test` and `after:integration-test` which would not be defined in any lifecycle, but instead be dynamically created by virtue of specifying an execution bound to that phase.

```
<plugin>
  ...
  <executions>
    <execution>
      <id>start-server</id>
      <phase>before:integration-test</phase>
      <goals>
        <goal>start</goal>
      </goals>
      ...
    </execution>
    <execution>
      <id>stop-server</id>
      <phase>after:integration-test</phase>
      <goals>
        <goal>stop</goal>
      </goals>
      ...
    </execution>
    ...
  </executions>
  ...
</plugin>
```

Every phase in any lifecycle would have its own implicit `before:` and `after:` phases in the lifecycle.

The logic of using `:` in these prefix names is that it would expressly be impossible to invoke these dynamic pseudo phases from the CLI as Maven will interpret any attempt to invoke them as `$plugin:$goal` and look for a `maven-before-plugin` or `maven-after-plugin`

The within phase ordering will be achieved by the addition of a `[`*`$priority`*`]` suffix. The priority will be an integer (positive and negative allowed) and execution of the lifecycle phase will invoke all bound MOJOs in order starting with the highest priority and ending with the lowest priority. Where two executions have the same priority, they will be executed in `pom.xml` order.

The logic of using `[]` in these suffix priorities is that these characters are in the typical reserved set for POSIX shells and used to specify a range of characters, again making it difficult to envision invoking a phase with priority from the command line without careful escaping. In any case the CLI would not permit the execution of a phase with priority. The CLI will only be able to execute a phase as a whole.

```
<packaging>war</packaging>
...
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <executions>
    <!-- create the jar file to use inside the war -->
    <execution>
      <phase>package[-1000]</phase>
      <goals>
        <goal>jar</goal>
      </goals>
      ...
    </execution>
    ...
  </executions>
  ...
</plugin>
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <executions>
    <!-- create the distribution that includes the war and docs -->
    <execution>
      <phase>package[2000]</phase>
      <goals>
        <goal>assemble</goal>
      </goals>
      ...
    </execution>
    ...
  </executions>
  ...
</plugin>
```

In order to allow lifecycle bindings per project type to include information about the pseudo phases and phase priorities, we would need to modify the syntax of the bindings reference, e.g. https://maven.apache.org/ref/3.6.2/maven-core/default-bindings.html

This proposal would modify the bindings XML schema to include optional attributes of `execution-point` and `priority` if not specified then the execution point would be considered to be the phase itself and not `before` or `after` and the priority would be assumed to be `0`

```
    <phases>
      ...
      <integration-test execution-point="before">
        ...:...:...:start
      </integration-test>
      <integration-test execution-point="after" priority="1000">
        ...:...:...:stop
      </integration-test>
      ...
    </phases>
```

The rationale is that this schema change is backwards compatible with the existing bindings schema and thus existing extensions defining bindings will still remain valid (just not able to bind to these dynamic phases)

We would also need to modify the `@Mojo` annotation adding new properties `executionPoint = ExecutionPoint.BEFORE|ExecutionPoint.AFTER` and `priority = <int>`

Lifecycle simplification

This proposal would lay the groundwork for the simplification of the Maven lifecycles. This proposal would only bring us to the transitional state, with the migration to the future state likely part of the Maven 5.0.0 effort.

Clean lifecycle

Current	Transitional	Future
pre-clean	~~pre-clean~~ (deprecated with recommendation to use *before:*clean)	use *before:*clean
clean	clean	clean
post-clean	~~post-clean~~ (deprecated with recommendation to use *after*:clean)	use *after*:clean

Default lifecycle

Current	Transitional	Future
validate	validate	validate
initialize	initialize	initialize
generate-sources	~~generate-sources~~ (deprecated with recommendation to use *before*:sources	use *before*:sources
	sources	sources
process-sources	~~process-sources~~ (deprecated with recommendation to use *after*:sources	use *after*:sources
generate-resources	~~generate-resources~~ (deprecated with recommendation to use *before*:resources	use *before*:resources
	resources	resources
process-resources	~~process-resources~~ (deprecated with recommendation to use *after*:resources	use *after*:resources
compile	compile	compile
process-classes	~~process-classes~~ (deprecated with recommendation to use *after*:compile	use *after*:compile
generate-test-sources	~~generate-test-sources~~ (deprecated with recommendation to use *before*:test-sources	use *before*:test-sources
	test-sources	test-sources
process-test-sources	~~process-test-sources~~ (deprecated with recommendation to use *after*:test-sources	use *after*:test-sources
generate-test-resources	~~generate-test-resources~~ (deprecated with recommendation to use *before*:test-resources	use *before*:test-resources
	test-resources	test-resources
process-test-resources	~~process-test-resources~~ (deprecated with recommendation to use *after*:test-resources	use *after*:test-resources
test-compile	test-compile	test-compile
process-test-classes	~~process-test-classes~~ (deprecated with recommendation to use *after*:test-compile	use *after*:test-compile
test	test	test
prepare-package	~~prepare-package~~ (deprecated with recommendation to use *before*:package	use *before*:package
package	package	package
pre-integration-test	~~pre-integration-test~~ (deprecated with recommendation to use *before*:integration-test	use *before*:integration-test
integration-test	integration-test	integration-test
post-integration-test	~~post-integration-test~~ (deprecated with recommendation to use *after*:integration-test	use *after*:integration-test
verify	verify	verify
install	install	install
deploy	deploy	deploy

To be determined:

- Do we really need a differentiation between sources and resources. If we have priority could we not just assign a different priority to the resource element of the sources leaving the default lifecyle in the future state as: `validate, initialize, sources, compile, test-sources, test-compile, test, package, integration-test verify, install, deploy` ?
- Do we need a differentiation between `validate` and `initialize` ?
- Should we add a special phase to represent all lifecycles, e.g. `before:*` that will always execute before any lifecycle starts and `after:*` that will always execute after any lifecycle completes?

Site lifecycle

Current	Transitional	Future
`pre-site`	~~`pre-site`~~ (deprecated with recommendation to use *`before:`*`site`)	use *`before:`*`site`
`site`	`site`	`site`
`post-site`	~~`post-site`~~ (deprecated with recommendation to use *`after`*`:site`)	use *`after`*`:site`
`site-deploy`	`site-deploy`	`site-deploy`