Lifecycle

The build lifecycle implemented for the first alpha has been extremely helpful, but there are some remaining tasks to allow it to be a complete replacement for the goal graph of Maven 1.x.

Defining the lifecycle according to packaging

I think it is worthwhile defining the lifecycle by the packaging, so the phases are fixed, but their mapping to goals are defined as configuration inside the artifact type handler. For example, this allows a plugin artifact to bind the plugin:descriptor goal automatically. It also allows a packaging of pom to not register any of the goals other than install/deploy.

The alternative is to always register them, disregarding the type, and use the type as a mojo execution guard - but I think this might be less clean in this particular case.

Solution

For per-packaging lifecycles, we can do this with plexus configuration, so a PackagingHandler can specify a lifecycle as configuration, but there will be a default (which is what is there now). It is a replacement lifecycle, not an overlay.

Lifecycle execution

I think the best solution for executions was:
@execute phase="..." lifecycle="some ID"
where phase should be replaced by a name that represents whatever we call phases/goals as a combination (task?)
the lifecycle ID will be inside a /META-INF/maven/lifecycles.xml file

- we went for a separate xml file instead of java code to be able to easily visualise it
- the lifecycle in here is overlaid on the existing one when the forked goal is called
- I've decided to aggregate them in one file for simplicity rather than having per mojo files
- I don't think we can use plexus configuration here because it would require an assumption in AbstractMojo

Lifecycle Use Cases

The proposed changes are to allow a parallel lifecycle to execute for certain goals, gathering the end results but not mixing in the lifecycle elements and changed parameters with the existing lifecycle.

So far, we have the @executePhase tag which does this - eg in idea:idea to execute generate-sources first.

Here, we look at a few use cases and how they might work with the proposed changes.

Use cases:

1. plexus application use case
2. idea:idea running generate-sources (defined by the mojo)
3. clover running tests
4. jcoverage, modifying class files
5. reports running tests, etc

Plexus application

want to be able to run "plexus:app plexus:bundle-application plexus:test-runtime"

plexus:app and plexus:bundle-application are analogous to war:exploded and war:war, so it would be better there for:

1. app and bundle to share the same logic, so app is run for a quick deployment, but bundle is all that is actually needed
2. plexus:bundle-application be the goal used for "package" when type = plexus-application

plexus:test-runtime is like making an assembly. "package" would be the executePhase for this goal, unless it were intended to be attached like an assembly is (though the assembly plugin is probably capable of doing that on its own)

idea:idea running generate-sources

This is covered by the current addition of @executePhase to the mojo declaration, so that generate-sources is always run before the execution of idea:idea as a new lifecycle instance, with the results that affect the project (ie the addition of a new compile source root) still applies afterwards.

The project used needs to be a clone that lasts the life of the executed lifecycle and the idea:idea goal, but goes back to normal afterwards so the effects don't hit other goals.

Clover use case

M1 plugin:

- clover:init -> sets up classpath, no real work
- clover -> clover:test, clover:report
- clover:on -> sets up clover compiler, turns of test failure, turns on test forking, changes class dest
- clover:off -> undoes clover:on
- clover:test -> on, test, off (builds database)

Desired use: m2 clover:clover

1. register clover compiler goal into generate-sources
2. modify output directory
3. run test (based on executePhase)
4. report database later reused

register clover compiler goal into generate-sources

- the binding is a parameter of the executed goal, not any existing lifecycle, so is different. Don't want to use @phase
- declaring this as part of the qdox block could be a bit tricky - we'll use XML.
- this is an overlay, just like the type handlers, so reuse that

modify output directory

- in this specific case, we actually want to keep it all separate, so we are adding a new output directory to a source root, so we need to pair those up, and also add a new runtime classpath element

run test

- as for other executePhase above
- can't do any dependendant stuff in clover:clover itself, as executePhase is done before that is executed

How to bind the clover compiler goal:

@goal clover
@executePhase test

META-INF/maven/lifecycle.xml (one per plugin - could it be in plugin.xml?)

```
<lifecycle>
  <phase>
    <id>generate-sources</id>
    <goals>
      <goal>
        <id>clover:compile</id>
      </goal>
    </goals>
  </phase>
</lifecycle>
```

```
@goal compile
@parameter name="outputDirectory"
expression="$\{project.build.directory\}/generated-sources/clover"
```

Jcoverage use case

Different to clover in that it modifies classes rather than adding compiled classes. Still not looking at the reporting case just yet.

Desired use: m2 jcoverage:jcoverage

1. register jcoverage:instrument goal for process-classes
2. modify classpath for test running
3. run test
4. report database later reused

register jcoverage:instrument goal for process-classes

- done with an overlay just like clover

modify classpath for test running

- we use some configuration in the lifecycle overlay which specifies new defaults
- the ognl symmetry could play some part in this instead, so depends on how John works that.
  eg, instrument could do @export outputDirectory project.build.outputDirectory

run test

- as for other executePhase above

How to bind:

```
@goal jcoverage
@executePhase test
```

META-INF/maven/lifecycle.xml (one per plugin - could it be in plugin.xml?)

```
<lifecycle>
  <phase>
    <id>process-classes</id>
    <goals>
      <goal>
        <id>jcoverage:instrument</id>
      </goal>
    </goals>
  </phase>
  <!-- ... -->
  <phase>
    <id>test</id>
    <goals>
      <goal>
        <id>surefire:test</id>
        <configuration>
          <!-- This assumes this is used instead of adding a runtime classpath
element, which might be a good idea -->

<classesDirectory>${project.build.directory}/generated-classes/jcoverage</classesDirec
tory>
          <ignoreFailures>true</ignoreFailures>
        </configuration>
      </goal>
    </goals>
  </phase>
</lifecycle>
```

```
  @goal instrument
  @parameter name="outputDirectory"
  expression="$\{project.build.directory\}/generated-classes/jcoverage"
```

Other reporting

Do other reports require anything additional?
junit-report:

- this executes the test goal, no change in parameters

pmd:

- standard report
- will use generated sources so needs to ensure this has been run
- might also fail based on certain threshold
- all fits

overall site:

- is this a new lifecycle, or does it run each goal on its own based on the registered reports? I believe it is the latter.

Random notes

- does executePhase need something that indicates "pre" in the name?
- did we decide on something like an around notation AOP style for the lifecycle?

- do we really execute a lifecycle, or just require to avoid re-executing?
    - this would avoid running test:test again
    - theoretically, test:test should be smart enough not to run again anyway
    - this could get back to the @prereq situation to some extent
    - how is it actullay run if it hasn't been already? forked lifecycle? Is this a little inconsistent?
    - common use is generated-sources which might not be fast to run over and over
    - here I would like to flatten down the lifecycle
    - still means getting back to a @requires style notation, and auto-ordering seems dangerous
    - perhaps it isn't doing any ordering? All the reports are in a bucket that are in the right place, and only required phases get activated?
    - or does it start from a blank lifecycle and each additional goal overlays additional stuff?
    - does this cause a risk of things not working because there are now three test executions, each with different parameters?
    - can we instead do the forked execution, but make use of the execution strategies so things still only execute once with identical parameters?
    - but what if another goal modifies some of its parameters?
    - general behaviour is that if something is specified twice now, it is run twice, because of the removal of prereqs
    - inclined to say we do the forking here for now at least.
- @phase vs. lifecycle declaration is a bit arbitrary. eg, should resources:resources always be part of process-resources? Why not use the @phase and somehow trigger it's inclusion?
- encapsulate all transfer in mojo fields, use OGNL to translate back to project
    - how would this be specified? does each field that exports provide an @export tag, or does it have an in, out, inout specifier? (urgh, CORBA flashback!)