Grouping

by Ovidiu Predescu, Jeff Turner

1. Grouping

Groups are like containers for Anteater objects, allowing reuse of definitions:

```
<group id="mygroup">
  <session/>
  <logger type="xml"/>
  <property name="host" value="localhost"/>
</group>

<!-- Each member task inherits the group's logger and session -->
<httpRequest group="mygroup" path="/a.html" .. />
<httpRequest group="mygroup" path="/b.html" .. />
```

If a task (like httpRequest) is part of a group, then it automatically uses whatever objects belong to that group.

1.1. Group Properties

Anteater tasks' behaviour is configured through properties of the group to which the task belongs. Currently recognised properties are `host`, `port`, `debug`, `timeout`, `protocol`, `haltonerror`, `enable` and `usetidy`. So if we had:

```
<group id="cocoontests">
  <property name="host" value="myhost.com"/>
  <property name="port" value="8080"/>
  <property name="debug" value="0"/>
</group>
<httpRequest group="cocoontests" ... />
```

```
<httpRequest group="cocoontests" ... />
```

Then those tasks would run against myhost.com:8080, with debug level 0, unless overridden by attributes on the httpRequest object. See the Configuration section for more details.

Group properties can also be set from outside a group:

```
<property name="cocoontests.debug" value="2"/>
```

This allows group values to be specified in properties files outside the test script, or from the command-line, eg:

```
anteater -Dcocoontests.host=localhost -Dcocoontests.debug=2 -f tests.xml
```

1.2. Group Inheritance

Since a Group object is an Anteater object, a Group can belong to another Group, either by nesting:

```
<group id="a">
  <property name="host" value="myhost.com"/>
  <group id="b"/>
</group>
```

or by the `inherits` attribute (`group` also works):

```
<group id="a">
  <property name="host" value="myhost.com"/>
</group>
<group id="b" inherits="a"/>
```

Group elements are inherited in what I hope seems a natural manner. Properties are passed through unless overridden, so `b` in the above example has host `myhost.com`. Loggers are passed through, unless any loggers are defined in the child group. Same with sessions.

1.3. The Default Group

There is an implicit `default` group, to which all tasks belong unless otherwise indicated. If the default group were written out, it would look like this:

```
<group id="default">
  <session/>
  <logger type="colour"/>
  <property name="host" value="localhost"/>
  <property name="debug" value="0"/>
  <property name="port" value="BUILTIN,8080"/>
  <property name="timeout" value="30s"/>
  <property name="protocol" value="HTTP/1.0"/>
  <property name="haltonerror" value="false"/>
  <property name="usetidy" value="false"/>
  <property name="usetidy-server" value="false"/>
  <property name="filename-format" value="true"/>
  <property name="overwrite" value="true"/>
  <property name="enable" value="true"/>
```

```
  <!-- Declare all other groups as children of 'default' here -->
  <group refid=".."/>
  ...
</group>
```

So by default, all tasks get a session, and a logger that prints to stdout, plus a bunch of properties used to configure the default Anteater behaviour.

The default group can be augmented and modified by the user, by declaring a group with id `default`. This way, we can override specific properties for all tasks:

```
<group id='default'>
  <property name="host" value="myhost.com"/>
  <property name="port" value="8080"/>
</group>
```

Or add another logger for all tasks:

```
<group id='default'>
    <logger type="xml"/>
</group>
```

All other items are inherited from the `default` defaults.

And of course the `default` group properties can be overridden at the command-line, e.g. `-Ddefault.host=myotherhost.com` or `-Ddefault.debug=10`.

1.4. Putting it all together

The purpose of grouping has been to make simple things easier, and complicated things possible. Some scenarios, from simple to complex:

1.4.1. Simple grouping

With the advent of the default group, most users need never bother with loggers, sessions, groups or properties. They just rely on the defaults, maybe occasionally overriding them, e.g. `-Ddefault.debug=5`.

1.4.2. Moderate grouping

For users for whom the defaults need modifying, that can easily be done by overriding the `default` group, and otherwise not touching the script. Want to log to XML as well as the console? Redefine the default group:

```
<group id="default">
  <logger type="minimal"/>
  <logger type="xml" todir="${log.dir}"/>
</group>
```

1.4.3. Advanced grouping

Users with somewhat large scripts, who want to break it up into sections can do so, by defining a hierarchy of groups:

```
<project name="groupdemo" default="main">
  <taskdef resource="META-INF/Anteater.tasks"/>
  <typedef resource="META-INF/Anteater.types"/>

  <group id="mytests">
    <property name="debug" value="0"/>
  </group>
  <group id="livesite" inherits="mytests">
    <property name="host" value="www.mysite.com"/>
    <logger type="xml" todir="{docs.dir}"/> <!--
    HTML report -->
  </group>
  <group id="devsite" inherits="mytests">
    <property name="host" value="www.mysite-dev.com"/>
    <property name="debug" value="1"/> <!-- devsite a bit unstable -->
    <property name="failonerror" value="true"/> <!-- Don't waste time testing whole sit

    <group id="devsite-brokenbit"> <!-- Very broken bit of devsite -->
      <property name="debug" value="10"/> </group>
  </group>

  <target name="main">
    <!-- Will have debug=10, host=www.mysite-dev.com, failonerror=true, and log
    to the console -->
    <httpRequest group="devsite-brokenbit" path="/broken.html"/>
  </target>
</project>
```

So we define a hierarchy of groups at the top of the script, and then use it in the subsequent tests.