Build Restructuring Proposal 10/30/2008

There are two primary goals of this effort:

- restructure so that our full build does not call targets multiple times (as is currently done, this is causing unpredictability in the builds)
- standardize calling conventions, dependency definitions, standard targets and variables (to make maintenance easier).

This effort is a clean-up, not a redesign. I am also looking at a complete redesign, but our current need is too high and the turn-around to redesign too long to initiate now.

The following have been identified as requirements for the build system in general (let me know if you have others):

1. Must be able to generate a full build from a svn checkout.
2. Must be able to generate a class level compile of a component from an standard svn check-out, if all the dependencies are available.
3. Should be able to generate a class level compile of a component from a project checkout from svn (org.eclipse.persistence.core on the same level as org.eclipse.persistence.moxy)Build Laws
4. Should build as efficiently as possible (The faster the build the more tesing we can do):
   - ContinuousBuild needs to complete within 30 minutes
   - Nightly needs to complete within 90 minutes

Below is the proposal for the Standards with regards to the Full build, Component Builds, and General Standards. For clarity, I'll define the automated build (it is not covered by this proposal).

<u>The Automated Build</u>

Scope is orchestrating the various automated processes:

- It does not compile or package any deliverables (calls to the full build to do that).
- It defines cb, nightly, milestone and release processes
- It generates automated web content
- It uploads appropriate deliverables to download site.

<u>The Full Build (trunk/build.xml)</u>

Scope is the generation of integrated (cross-component) deliverables:

- knows where to go to build components and tests
- does not define behavior on how to build component parts (dependency definitions, jar names, etc).
  - this insures that the main build needs no info on how to build or what tasks to perform.
- does define behavior for integrated deliverables: javadocs, eclipselink.jar
- will only call the "all" behavior of a component:
  - means fewer ant calls, so build should be faster.
  - means that there could be fewer targets in build:

            - will still contain “upper-level” targets to:
                - only build components, main jar, and tests
                - build oracle extentions (after the above)
                - build full distribution
                - build all tests
                - run srgs
                - run lrgs
            - won’t need separate “clean”, “build”, “jar” steps anymore
            - will keep "clean", it is nice to be able to clean-up the full "built" tree, but:
                - "clean" will not be called by any of the upper-level targets (see component build)
                - will create “clean-local” to clean-up the "full build" working areas and generated files
            - the "build-core" target will change to "build-components", but won't execute "oracle-extensions"
            - "package-eclipselink" will depend upon "build-components", and create the jar from the other component bundles
            - build-util will encapsulate build-workbench, the package rename build, and util-dbws
            - package-deliverables to group package-installer, package-bundles, package-source, generate-javadoc, package-installer
    - All component calls will be in the form:
      `<antcall file="build.xml" dir="${component.dir}" target="${target}" inheretAll="true"/>`
        - insures that the component call is executed from its own directory context
        - guarantees that basic variables set earlier are retained (like version.string)
        - means that heirarchy variables need to be unique for each component, and full build
            - The only exceptions will be the class.destination.dir, jar.destination.dir and relative.build.dir directory variables

<u>Component build files</u>

Scope is the cleaning, compiling, packaging, and sometimes testing of a component:

- only know about themselves (the rest of the build is a black box).
- "${user.home}/user.properties" will be loaded just after the project is defined
    - allows over-ride of otherwise generated variables
    - ${user.home} is not guaranteed to be populated on all platforms, but in the case it isn’t can be defined on command-line
    - is descriptive, and can be easily found if the name of the file needs to be changed
- all component variables will be component unique (core.compile.path, core.jar.name, etc)
    - will avoid inadvertently using another components compile.path, etc
- will use a common test methodology to determine flat vs tree hierarchy
- will set ${"component".build.tree.is.flat} and test it to generate other "component" variables
- will populate their relative dependency locations to a single variable for each in either case without the use of “.” or “..”
  ${jpa.plugins.dir} = "plugins" for flat or "jpa/plugins" for tree
  ${common.plugins.dir} = "plugins" for flat or "plugins" for tree
  ${relative.build.dir} = ".." for flat or "../.." for tree (set to "." in full build)
- create their classes and assemble jars to common variable:
    - simplifies the build.
        - allows full build to go to a common location for classes or jars.
        - components can test a single location for all eclipselink dependencies
    - class.destination.dir will be generated to "${relative.build.dir}/classes" (flat or tree) or "classes" for full build.
    - jar.destination.dir will be generated to "." (flat or tree) or "jars" (full build).
- test for all dependency existence, fail if not found (do not attempt to build them).
- define 4 (5) behaviors: all, clean, compile, package, (test)
    - none of the tasks (except "all") should have built in dependencies

- The "all" target is the first target line after the project definition:
  `<target name="all" depends="clean, compile, package" description="build Core component"/>`
- the "package" task will only execute if "flat" is defined

<u>General Standards</u>

- All ant "project" names will be the name of the dir they are in:
  - org.eclipse.persistence.jpa
  - eclipselink.moxy.test
  - is descriptive and unique
  - Allows antfiles to easily generate where they should be based upon who they are
    - basedir may not be set as expected depending upon ant version and how file is called
- All jar variable definitions (product or 3rd-party dependency) will:
  - only contain the jar name (no paths)
    - allows the path calculations to work
  - end in ".jar"
    - makes build easier to follow:
      destination="${jar.destination.dir}/${jpa.bundle.jar}"
- All directory variable definitions will:
  - not contain beginning or ending slashes
    - arbitrary standard, but I think it is easier to read as well
  - end in ".dir"
    - makes build easier to follow
    - dependency paths would then be constructed as:
      ```
      dir="${relative.build.dir}/${common.plugins.dir}"
      file="${relative.build.dir}/${jpa.plugins.dir}/${jpa.prototype.jar}"
      dir="${class.destination.dir}"
      ```
- Test build.xml files will:
  - Not have "build" as a dependency of "run" targets (Many build types need to build the tests for varification)
  - Contain only contain non-server targets
    - separation of the two makes maintenance easier
      - server type tests should be stored in a separate xml (server-tests.xml)
    - we do not want duplication of compiles
      - LRGs and SRGs do not run on a "server" so don't need to be compiled as part of execution, but server tests need to recompile for each server they are run on.
  - SRG and LRG tests will run against jars
  - "local" run will be against classes
  - contain "all" like a standard component (clean, compile, jar)
  - contain "run" targets for test execution
    - for example "run-cmp", "run-lrg", "run"