Part Handling in qooxdoo - II

Part Collapsing

Thomas Herchenröder, 1&1 Internet AG (November 26, 2013)

Part Collapsing

Parts are collapsed, i.e. made smaller, by *merging packages* so that each part is made up of *fewer* of them. Merging of packages is an iterative process that is constraint by various rules. Merging is currently done in two phases

- by load order
- by size

When merging, a *target package* receives the *source package*'s classes, the source package is then removed. The phases mainly differ in how source packages are *selected*.

Package Merging

This is the basic process to merge one package into another. Let p_s be the source and p_t the target package, then

- within a part, start with the *last* package in the (dependency-sorted) package list
- iteratively check each package further up the list if it is a suitable merge target
- if found, add $Classes(p_s)$ to $Classes(p_t)$, maintaining load ordering within the target package
- add $Deps(p_s)$ to $Deps(p_t)$; this is a **recursive** process and might incur new packages being added to certain parts
- remove p_s from all parts
- repeat with the next package from the end of the list (whether or not the previous package could be merged), until all packages have been tried

Source Package Selection

The two merge phases (*load order* and *size*) refer to the way source packages are identified.

- In the *size* phase, packages are selected by size, i.e. packages below a configurable threshold are set up for merging. This is to avoid packages that are too small.
- The selection in the *load order* phase is more complicated (see later). The aim is to have fewer packages to load.

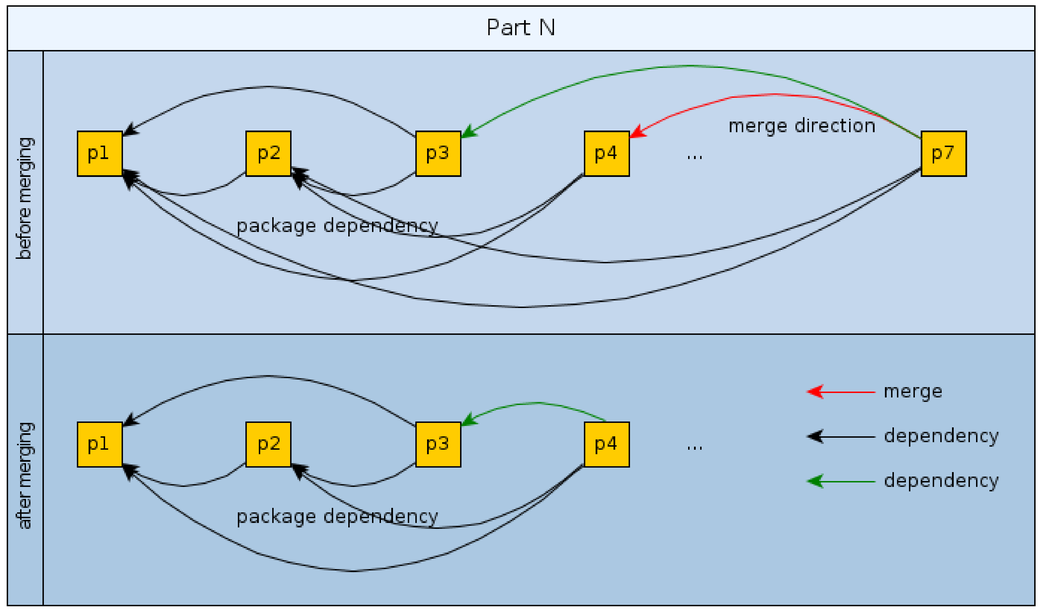

Figure 1: Merging packages and maintaining dependencies

Target Package Selection

The following conditions have to be met for a package to be eligible as a merge target.

- $Parts(p_s) \subset Parts(p_t)$ [this assures the classes of p_s remain available wherever they are needed]

Merging by Load Order

The aim of load-order merging is to have just one additional package to load with each new part.

- By using the *expected-load-order* configuration key the user groups together parts (*load groups*) where the number indicates the expected load order.
- This allows for more aggressive package merging.
- As a corner case, a load group can consist of a single part.
- Without this configuration key, or within a single group, no load order is assumed.
- First, all *unique* packages within the load group are merged.
- Then, packages *common* among the parts of the load group are merged.
- Target packages in one load group become **closed** for subsequent groups. This is to avoid "monster" packages.
- The *boot* part is always load order-merged by default. It has load order number *0* which cannot be assigned to any other load group.
- Parts which are not in any load group are not actively collapsed. (They are still affected by package changes if packages they use are merged).

References

The following qooxdoo applications use parts, with and without explicit load-ordering:

[1] Feedreader Configuration, (load groups commented out),
https://github.com/qooxdoo/qooxdoo/blob/release_3_0_1/application/feedreader/config.js

[2] WidgetBrowser Configuration, (no load groups)
https://github.com/qooxdoo/qooxdoo/blob/release_3_0_1/application/widgetbrowser/conf...

[3] 3C MailClient Parts Configuration, (2 load groups)
https://svn.1and1.org/svn/ccclient/qooxdoo-client/trunk/conf/parts.json

Merging by Load Order (cont.)

Load-order merging **does not** destroy the ability to load parts *in arbitrary order*. But if parts are loaded out of order a larger number of unneeded classes will be loaded with the required packages.

Questions

- Is the condition for a target package trivially true for all "larger" packages in a part?
- Parts which don't belong to any load group are not processed in the load-order phase. Why?
- Whithin a load group, why are unique packages processed first, then common packages?
- Why are merge targets restricted to one use only?

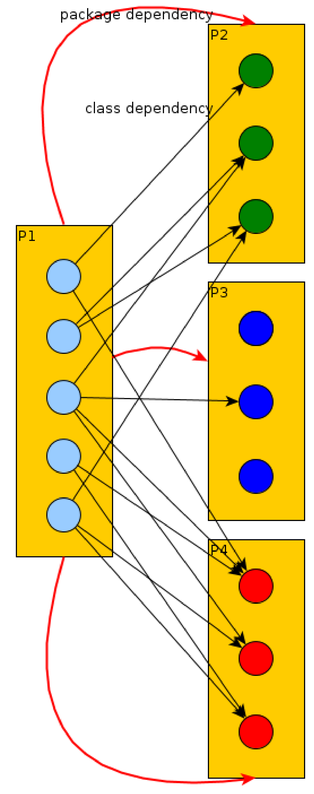

Figure 2: Class dependencies induce package dependencies