With these manifest files, we just needed a way to parse them and do something useful. I wrote a launcher program, imaginatively called "Launcher," to do just that.

Launcher

I've seen many desktop Java applications that come with huge shell or batch scripts to locate the JRE, set up environment variables, build the classpath, and so on. Ugh.

Given a module name, Launcher parses the manifest files, building the transitive closure of that module's dependencies. Launcher is careful not to add a module twice, and it resolves the set of partial orderings into a complete ordering. Figure 4-3 shows the fully resolved dependencies for `StudioClient`. `StudioClient` declares both `StudioCommon` and `Common` as dependencies, but Launcher gives it only one copy of each.

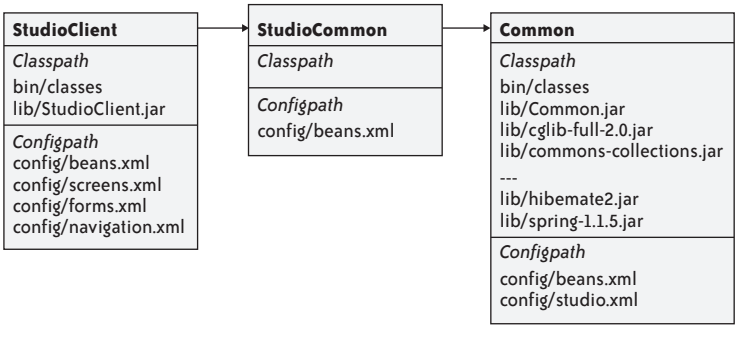

FIGURE 4-3. Resolved dependencies for StudioClient

To avoid classpath "pollution" from the host environment—ANT on a build box, or the JRE classpath on a workstation—Launcher builds its own class loader from the combined classpaths. All application classes get loaded inside that class loader, so Launcher uses that class loader to instantiate an initializer. Launcher passes the configuration path into the initializer, which creates all the application context objects. Once the application contexts are constructed, we're up and running.

Throughout the project, we refactored the module structure several times. The manifest files and Launcher held up with only minor changes throughout. We eventually arrived at six very different deployment configurations, all supported by the same structure.

The modules all share a similar structure, but they don't have to be identical. That was one of the side benefits of this approach. Each module can squirrel away stuff that other modules don't care about.