Aaron Schultz
Software Developer
National Center for
Ecological Analysis and Synthesis

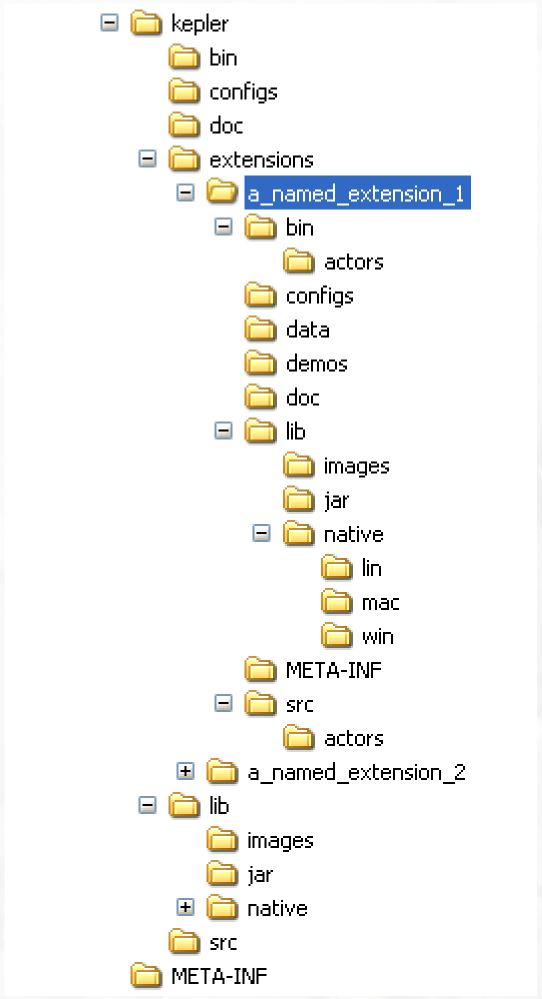
Dynamic Loading of Kepler Extensions

2008-06-04

Outline

- Standardized Directory Structure
- What happens at Run-Time?
- What happens at Build-Time?
- Installing Extensions
- Benefits
- Risks
- Reference

Standardized Directory Structure



```
⊟ kepler
    bin
    configs
    doc
  ⊟ extensions
    ⊟ a_named_extension_1
      ⊟ bin
          actors
        configs
        data
        demos
        doc
      ⊟ lib
          images
          jar
        ⊟ native
            lin
            mac
            win
        META-INF
      ⊟ src
          actors
    ⊞ a_named_extension_2
  ⊟ lib
      images
      jar
    ⊞ native
    src
  META-INF
```

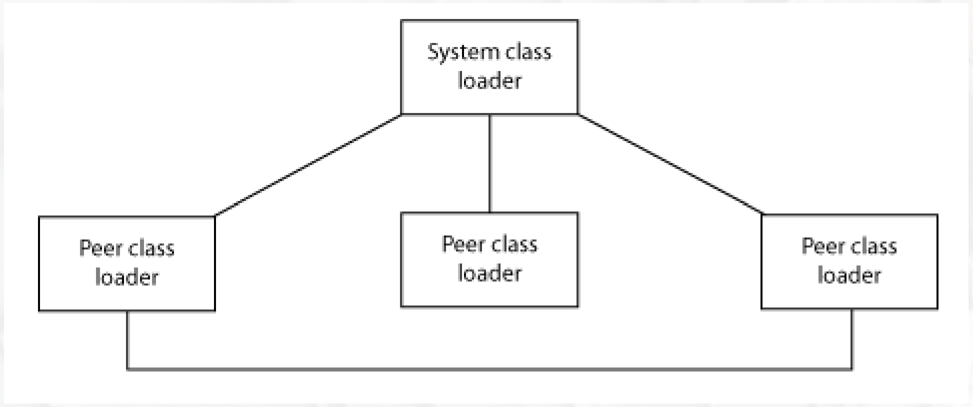
- An “extensions” folder under the kepler directory contains a folder for each extension.
- Extension source files are kept in the extension src directory and are compiled to the extension bin directory
- Each extension has a spcialized jar manifest and would be package as KARs
- Each extension has its own lib directory for images, jars, native libraries, etc.
- Each extension has its own demos and doc directory
- Any of these directories can be included or ommitted
  - For example, an extension of only actors could include only bin/actors

What Happens at Run-Time?

- At run-time, extension resources are detected by the ExtensionManager in the standardized Kepler extension directory structure (packaged or not)
- An instance of a custom ClassLoader is created for each extension.
  - Extension classes and jars run in their own namespace, solving jar dependency issues and Class naming conflicts (see next slide)
    - To achieve full jar dependency, class loader delegetion is restricted by package (not the full binary class name)
  - Extension classes have full visibility of Core classes and can override Core classes in their namespace without affecting the Core
- Core classes run in their own namespace and have no visibility of extension classes, except through system interfaces used as Core extension points
- To modify classes in the Core namespace override information in the manifest would request that a class be excluded from the core namespace and the extension class or jar be used instead (in effect the same as the exp directory now but would happen dynamically at run-time)
  - This mechanism can be used to allow extension developers to develop, test, and request changes to the Core
  - Conflicts between different extension Core overrides would be managed and handled before loading any classes

Peer Delegation Model

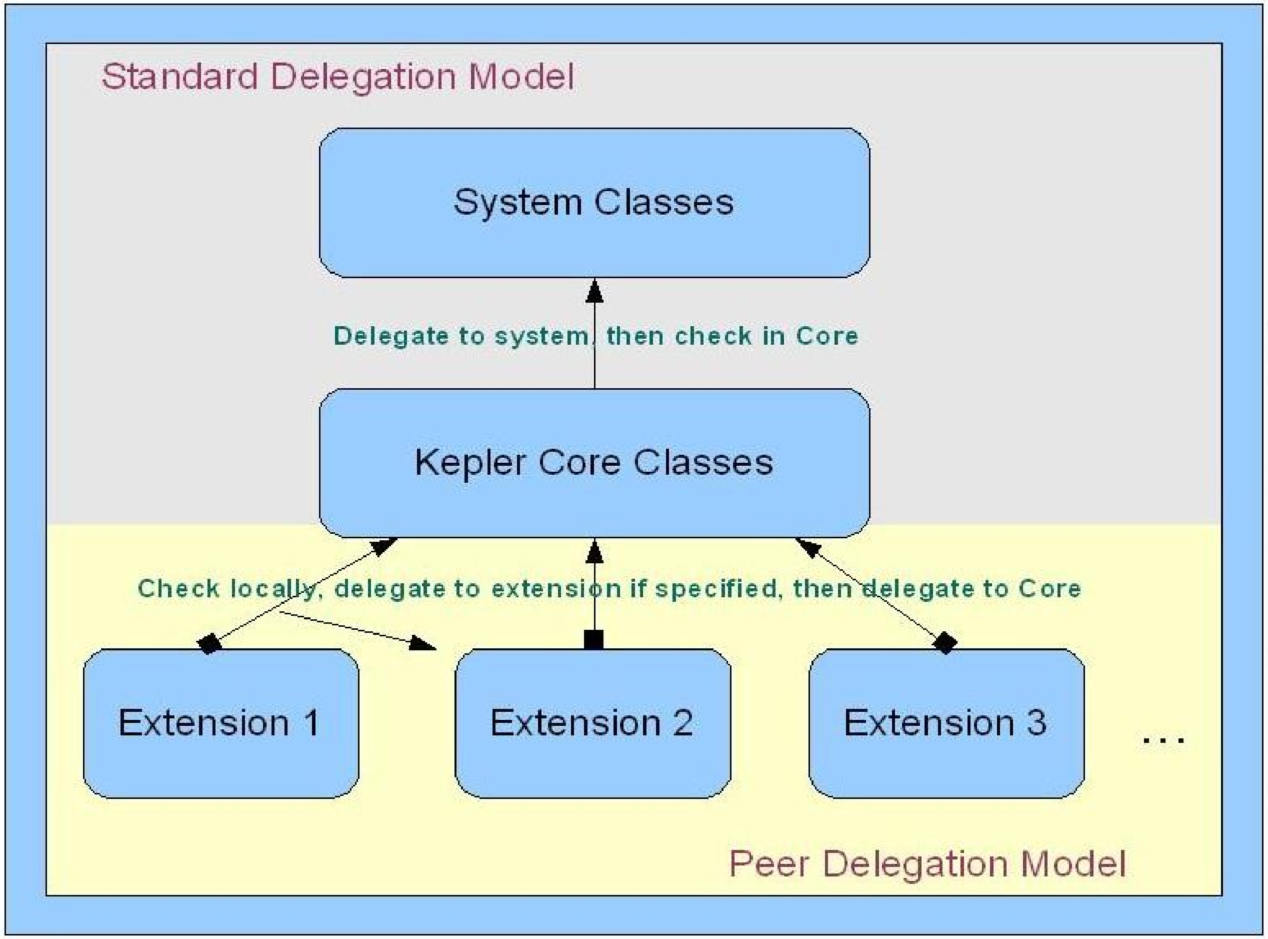
Peer class loading does not follow the traditional hierarchical delegation structure for class loaders. Instead, it has a set of class loaders that are unrelated except that they have the same parent (usually the system class loader). These class loaders can delegate not only to their parent, but also to their peers.



This kind of class loader structure allows discrete class spaces to exist within one JVM; thus, it's very useful for running componentized products. A good example of this class loading structure is an OSGi framework, such as the one Eclipse is built on.

From Demystifying class loading problems, Part 4

Kepler Implementation of Peer Delegation Model


Standard Delegation Model
System Classes
Delegate to system, then check in Core
Kepler Core Classes
Check locally, delegate to extension if specified, then delegate to Core
Extension 1
Extension 2
Extension 3
...
Peer Delegation Model

Separation of Kepler extension namespaces using ClassLoaders with the ability for extension to override Core classes at run-time

- Classes can not be unloaded from the Standard Java system classloader
- To facilitate dynamic extension override of Core classes a custom classloader must be used to load all Core classes.

What Happens at Build-Time?

- A utility is run once to auto detect the current build configuration and generate either an Eclipse project, a Netbeans project, or a set of Ant build files
- Each extension would have a Manifest that included dependency information, Core override information, version information/LSID, and licensing information for that extension.
- Extenders and developers use eclipse, netbeans or ant to build that configuration. If the configuration changes (i.e. The Manifest information changes), the utility would need to be rerun.
- This utility would use the same classes as the run-time system for determining the configuration from the extension manifests

Installing Extensions

- Initially, extensions could be installed by manual download and extraction into the extensions directory of a previously installed Kepler Core – restart of Kepler would auto-detect and load the new extension
- Later, custom menu options could be developed to automate the download of extensions followed by an automated restart
- Different “flavors” of Kepler could be developed by simply including various extension KARS in with the installer

Benefits

- This solution is very similar to several existing systems
  - Tomcat, Jrun, Eclipse, FireFox
- Flexibility and power to do whatever we need to do to make extensions work how we want them to work – few constraints
- Simple and easily understandable separation of extensions from Core (one extension, one directory/KAR)
- Smooth migration process from current state of the Kepler code and build process
- Will work well with NMI system
- Greatly reduced size to Core download
- Running extensions in their own Namespace relieves extension developers from worrying about conflicts with Core code or other extensions
- Extension developers can easily use their own code repositories

Drawbacks/Risks

- Run-time auto detection and loading may affect application startup speed
- A custom solution means developing and maintaining custom documentation of that solution
- Effort level for implementing this solution is unknown

References

- Java programming dynamics, Part 1 (IBM)
- Demystifying class loading problems Parts 1-4 (IBM)
- Dynamic Class Loading in the JavaTM Virtual Machine