Using CAL Archives (Car files)

Contributor: Joseph Wong
Last modified: June 13, 2007

Contents

1 Introduction

A CAL Archive – also called a *Car* file for short – is a file format based on the zip/jar file format and is used for aggregating files which constitute CAL resources. In particular, a Car file may contain:

- Workspace declaration files
- CAL source files
- cmi files (compiled module files)
- lc files (our custom-loaded lecc generated class files)
- metadata files
- gem design files
- user resource files

Car files are intended to be used as a deployment mechanism: instead of deploying CAL modules as a set of source files and resource files, these modules can be packaged into a small number of Car files, so that they can be distributed and deployed more easily. Section 5 contains a more in-depth discussion on various deployment strategies using Car files.

2 Cars, Vaults, and Workspaces

While a Car acts as a container of CAL resources, a Car itself is also a CAL resource. In particular, a Car is a *read-only* resource: the contents of a Car file cannot be modified at runtime.

As with other CAL resources, Car files are located in repositories known as *vaults*, such as the StandardVault (representing the Java classpath) or the SimpleCarFileVault (representing a standalone Car file).

On the other hand, resources contained within a Car are always fetched directly from the Car without going through the intermediate step of copying the uncompressed contents onto disk.

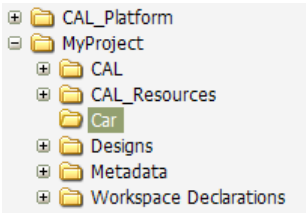
2.1 Cars and the StandardVault

There are two ways to use Car files with the StandardVault: 1) by putting the files in a particular directory under the regular StandardVault scheme, and 2) by adding *Car-jar* files (see Section 2.1.2) to the Java classpath.

2.1.1 Directory Structure

Similar to other types of resources such as CAL source files and workspace declaration files, Car files are grouped in their own directory for the StandardVault. This directory is named "`Car`" – note that this name is case sensitive: uppercase "`C`" and lowercase "`ar`".

The "`Car`" directory is the analog of the "`CAL`" directory for CAL source files and the "`Workspace Declarations`" directory for workspace declaration files. For example, the following figure shows a directory structure that includes some Car files stored in the appropriate place:



2.1.2 Using Car-jars with the Java classpath

As an alternative to putting Car files in a "`Car`" directory, one can add Cars to the StandardVault by putting them directly on the Java classpath. Unlike regular Car files, these variant Cars – called *Car-jars* – will have names ending with a `.car.jar` suffix, and will employ the Jar file format.

Even though a Car-jar file has a `.car.jar` suffix (e.g. `cal.platform.car.jar`), the `.jar` suffix is dropped for the actual name of the Car as recognized by the CAL runtime (e.g. `cal.platform.car.jar` is the Car-jar defining the Car `cal.platform.car`).

In terms of precedence, the Cars located in the StandardVault's regular directory structure take precedence over the Car-jars on the classpath.

The workspace declaration files contained within a Car-jar are recognized as files in the StandardVault, and thus can be referenced and imported in the same way as other workspace declaration files occurring in "`Workspace Declarations`" directories.

2.2 Importing a Car in a Workspace Declaration

To reference a Car in a workspace declaration, include a line:

```
import car StandardVault test.car main.carspec
```

where `StandardVault` refers to the vault where the Car file could be found, `test.car` is the name of the Car, and `main.carspec` is the name of the *Car Workspace Spec* file within the Car (see Section 2.3).

2.3 Car Workspace Spec files

A Car Workspace Spec (or Carspec for short) is a whitespace delimited list of module names, either representing all the modules in the Car, or just a proper subset thereof. The Carspec is the unit for importing modules from a Car into the workspace.

A Car file may contain more than one Carspec file. Many of the tools for building Cars default to using the name `main.carspec` for the Carspec file that represents all the modules in the Car.

2.4 Module resources in Cars

If a module is defined in a Car, then all its resources (e.g. the CAL source file, metadata files, user resource files) must also be encapsulated within the same Car. The runtime maintains a mapping from a module name to the corresponding location where resources for the module are to be found – either on disk or from a particular Car file.

In a workspace where all modules come from a set of Cars, no compilation should take place when the workspace is initialized, and there should be no lecc_runtime directory generated on disk, as all program runtime resources are fetched directly from the Car.

2.5 Sourceless modules in Cars

It is possible to deploy a Car file that does not contain any CAL source files. Modules that are deployed without source are called *sourceless modules*. These sourceless modules are loaded directly from the corresponding cmi files.

For these modules, the Car file will contain empty stub files in lieu of the original source files.

3 Building a Car

There are currently a number of ways to construct a Car out of an existing CAL workspace.

3.1 Building a single Car

One way to build Car files is to simply package up all the resources in the current workspace into a single Car.

3.1.1 In ICE

To package up the contents of the current workspace in ICE, use the `:car` command. For example, in ICE, one may type:

```
:car c:\temp
```

This will construct a new Car file `c:\temp\Car\ice.default.car` containing all modules and their associated resources in the current working workspace (i.e. `ice.default.cws`). Note that the Car file is generated under a subdirectory called `Car`. This is to facilitate the use of the generated Car directly in its output location (as described in more detail in Section 4).

Included in the Car will be a file named `main.carspec`, which is a Carspec file containing the names of all the modules.

Also generated along with the Car file will be a corresponding workspace declaration file that can be used to import the Car. This file, named `ice.default.car.cws`, will be located in `c:\temp\Workspace Declarations`.

The full syntax of the `:car` command is as follows:

```
:car [-keepsource] [-nocws | -nosuffix] [-s] [-jar] [-d] <output directory>
```

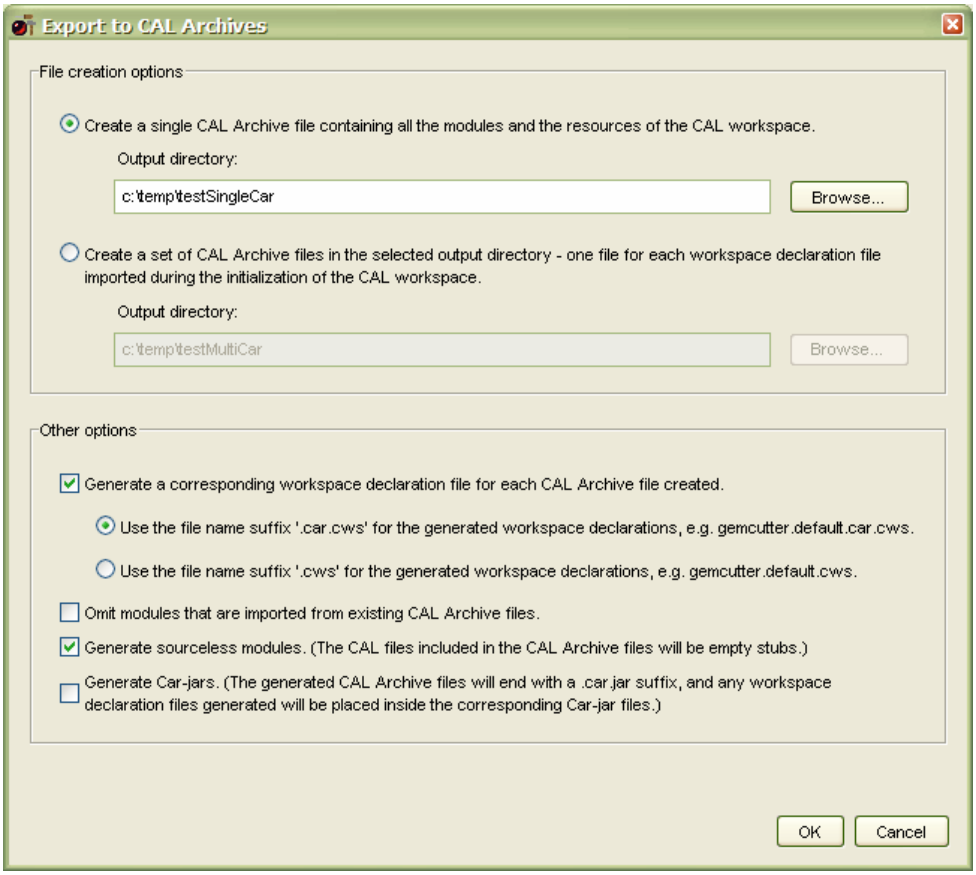
- By default, the Car is built with sourceless modules. If `-keepsource` is specified, then the generated Cars will contain the full source of the modules.
- If `-nocws` is specified, then a new workspace declaration file will not be generated for each output Car.
- If `-nosuffix` is specified, then the workspace declaration files generated will end simply with `.cws` rather than `.car.cws`.
- If `-s` is specified, then the modules that come from Cars will be skipped over in the output Cars.
- If `-jar` is specified, then Car-jars will be generated instead of regular Car files. In this case, the output files will be placed directly under the specified output directory, rather than in a `Car` subfolder. If `-nocws` is not specified, then the new workspace declaration file will be packaged *inside* the Car-jar.
- If `-d` is specified, then a Car file is created in the given directory for every workspace declaration file imported by the current workspace. (See Section 3.2.1)

3.1.2 In GemCutter

To package up the contents of the current workspace in the GemCutter, go to the Workspace menu and select the menu item "Export to CAL Archives…". From the options presented in the dialog, choose "Create a single CAL Archive file", then use the Browse button below to provide the name of the output directory.

Similar to the `:car` command in ICE, the new Car will contain all the modules and their resources in the workspace, and also a corresponding `main.carspec` file.

The options `-keepsource`, `-nocws`, `-nosuffix`, `-s` and `-jar` available in ICE are also accessible via the dialog in the "Other options" section:



3.1.3 Using the command line tool

The command line tool for building Cars is provided by the class `org.openquark.cal.services.CarTool`, and it can be invoked by creating an Eclipse run target on the class.

The command line syntax for this tool is as follows:

```
java org.openquark.cal.services.CarTool
 [workspaceDeclaration
 | -multi workspaceDeclaration... --]
 [-notVerbose] [-keepsource] [-nocws | -nosuffix] [-s]
 [-excludeCarsInDirs carName ... --] [-excludeCarJarsInDirs carName ... --]
 [-jar] outputDirectory [specFileName ...]
```

Where:

- *`workspaceDeclaration`* is the name of the workspace declaration file (e.g. ice.default.cws) which will form the basis of the modules that are included in the Car.
- If `-multi` is specified, then one Car will be generated per workspace declaration file appearing between `-multi` and `--`.
- If `-notVerbose` is specified, then additional diagnostic information will not be displayed.

- If `-excludeCarsInDirs` is specified, then the Cars found in the 'Car' subdirectories of the listed directories will not be generated.
- If `-excludeCarJarsInDirs` is specified, then the Car-jars found in the listed directories will not be generated.
- If `-jar` is specified, then Car-jars will be generated instead of regular Car files. In this case, the output files will be placed directly under the specified output directory, rather than in a `Car` subfolder. If `-nocws` is not specified, then the new workspace declaration file will be packaged *inside* the Car-jar.
- *`outputDirectory`* is the name of the output directory to which the single Car file will be generated (under a `Car` subfolder).
- *`[specFileName ...]`* is an optional list of additional Carspec files to be included with the Car. A `main.carspec` file is added to the Car by default.
- The options `-keepsource`, `-nocws`, `-nosuffix` and `-s` retain the same meaning as in the ICE `:car` command.

3.2 Building one Car per imported Workspace Declaration

Another approach to building Car files is to build a set of Car files from one workspace. In particular, we support the building of one Car file per workspace declaration file that constitutes the initial declaration of the CAL workspace.

For example, suppose we have the following workspace declaration files:

a.cws:

```
StandardVault Cal.Core.Prelude
StandardVault Foo
```

b.cws:

```
StandardVault Bar
StandardVault Baz
import StandardVault a.cws
```

c.cws:

```
StandardVault RunModule
import StandardVault b.cws
```

Suppose then that the CAL workspace is initialized with the declaration `c.cws`, we can build three Cars which correspond to the three workspace declaration files:

a.car: containing modules `Cal.Core.Prelude` and `Foo`

b.car: containing modules `Bar` and `Baz`

c.car: containing module `RunModule`

3.2.1 In ICE

To build one Car per workspace declaration file in ICE, use the `:car` command and specify the `-d` option. For example:

```
:car -d c:\cars
```

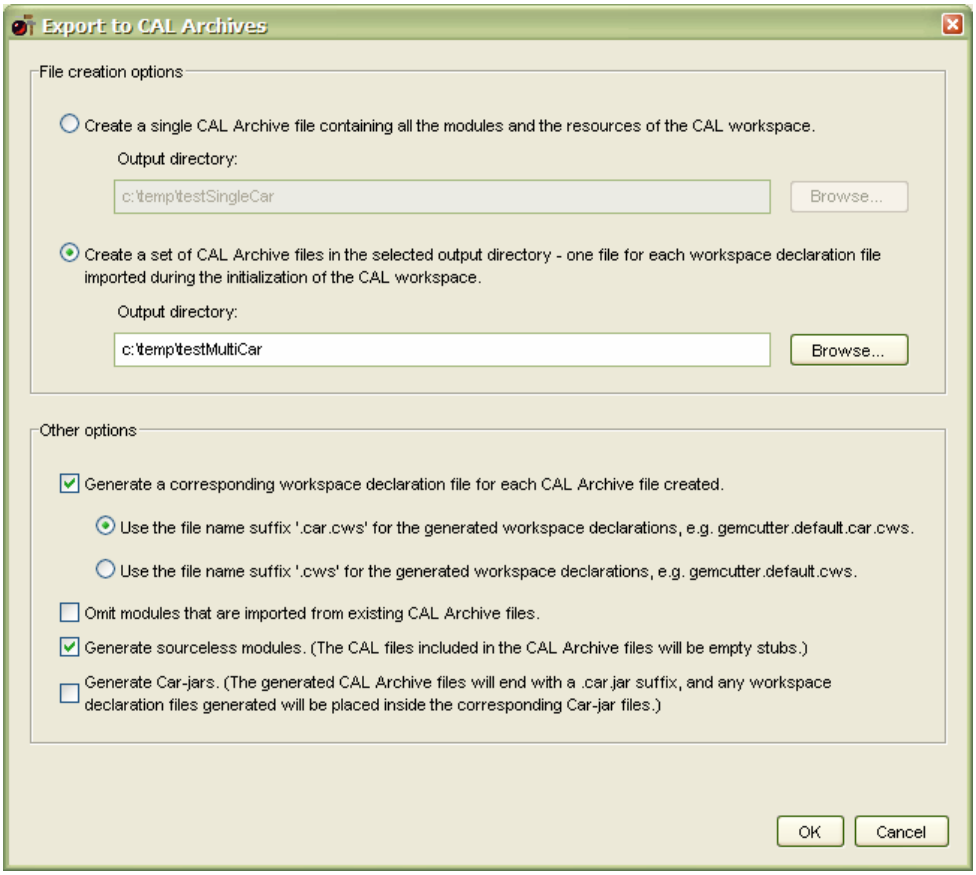
This will construct the set of Car files in the output directory `c:\cars`. Included in each Car will be a Carspec file named `main.carspec`, containing the names of the modules in that Car.

The command line options `-keepsource`, `-nocws`, `-nosuffix`, `-s` and `-jar` can also be used in conjunction with the `-d` option.

3.2.2 In GemCutter

To build one Car per workspace declaration file in GemCutter, go to the Workspace menu and select the menu item "Export to CAL Archives…". From the options presented in the dialog, choose "Create a set of CAL Archive files", then use the Browse button below to provide the name of the output directory.

For example:



3.2.3 Using the command line tool

The command line tool can be invoked with the `-d` option as well, using the following syntax:

```
java org.openquark.cal.services.CarTool
 [workspaceDeclaration | -multi workspaceDeclaration... --]
 [-notVerbose] [-keepsource] [-nocws | -nosuffix] [-s]
 [-excludeCarsInDirs carName ... --] [-excludeCarJarsInDirs carName ... --]
 [-jar] -d outputDirectory
```

Where:

- *`workspaceDeclaration`* is the name of the workspace declaration file (e.g. ice.default.cws). A Car will be built for this file, and also for each workspace declaration transitively imported by this file.
- If `-multi` is specified, then the generated Cars will be based on the set of workspace declaration files between `-multi` and `--`.
- If `-notVerbose` is specified, then additional diagnostic information will not be displayed.

- If `-excludeCarsInDirs` is specified, then the Cars found in the 'Car' subdirectories of the listed directories will not be generated.
- If `-excludeCarJarsInDirs` is specified, then the Car-jars found in the listed directories will not be generated.
- If `-jar` is specified, then Car-jars will be generated instead of regular Car files. In this case, the output files will be placed directly under the specified output directory, rather than in a `Car` subfolder. If `-nocws` is not specified, then the new workspace declaration files will be packaged *inside* the corresponding Car-jars.
- *`outputDirectory`* is the name of the output directory.
- The options `-keepsource`, `-nocws`, `-nosuffix` and `-s` retain the same meaning as in the ICE `:car` command.

4 Trying out Car Files in a Development Environment

One simple way to try using Car files in a development environment is to extend the StandardVault to include the generated Car files and workspace declaration files. There are two different methods to do so: using regular Car files, and using Car-jar files. (See Section 2.1)

4.1 Using regular Car files

The CAL runtime will recognize the Car files if the root output directory (i.e. the parent of the generated subdirectories "`Car`" and "`Workspace Declarations`") is added to the Java classpath.

For example, suppose one had generated some Car files in ICE using:

```
:car -d c:\testCar
```

Then, `c:\testCar` should be added to the *front* of the Java classpath.

Also, one needs to inform the CAL program in question (e.g. ICE or GemCutter) to use one of the generated workspace declarations so that the CAL workspace will be initialized with the contents of the Cars.

We will outline the entire process of setting up ICE and GemCutter in the Eclipse environment in the following sections.

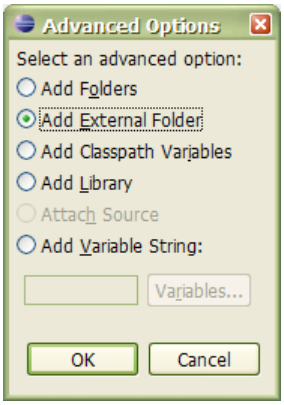
4.1.1 ICE

First, set up an Eclipse run target for ICE.

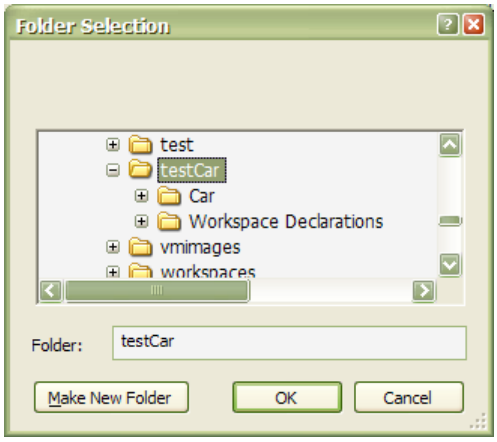
4.1.1.1 Modifying the Java Classpath

To add the generated Car files and workspace declaration files to the Java classpath, edit the properties of the run target and navigate to the Classpath tab.

Once there, select the "User Entries" tree node in the Classpath panel, then click the "Advanced…" button.

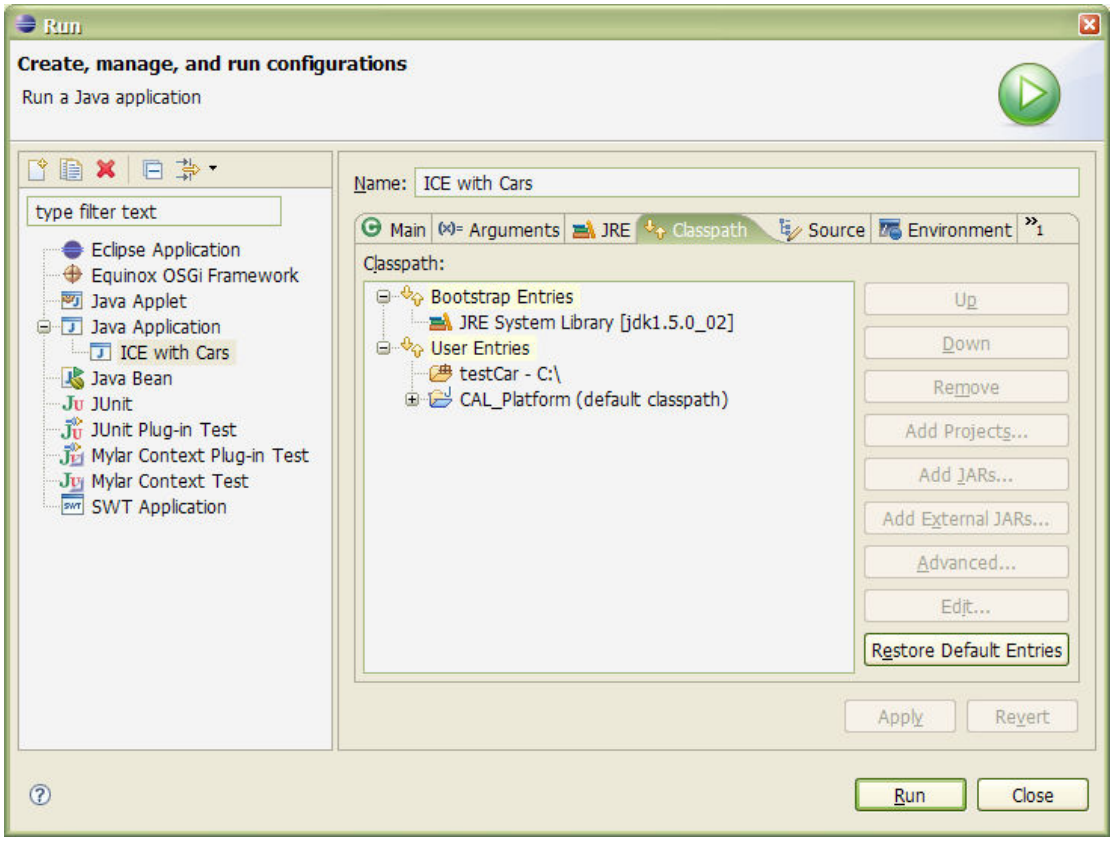


Select "Add External Folder" in the Advanced Options dialog, and click OK. A directory selector will then pop up, where the root output folder containing the "`Car`" and "`Workspace Declarations`" subfolders should be selected:



Afterwards, use the "Up" button to position this new classpath entry to the *top* of the "User Entries" section.

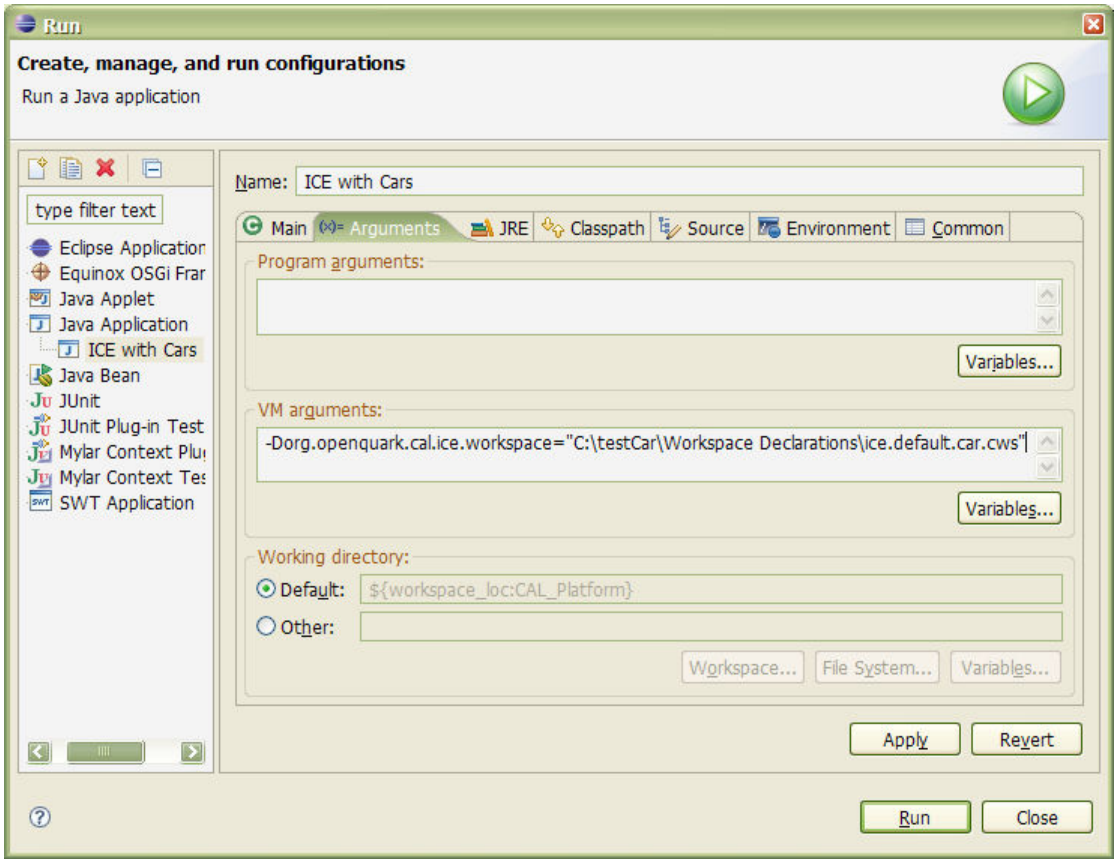
The final result should look like this:



4.1.1.2 Specifying an Alternate Workspace Declaration

Next, we need to tell ICE to use an alternate workspace declaration. This can be achieved by specifying the system property "org.openquark.cal.ice.workspace". The value of this property shall be the full path of one of the generated workspace declaration files, surrounded by double quotes.

For example:



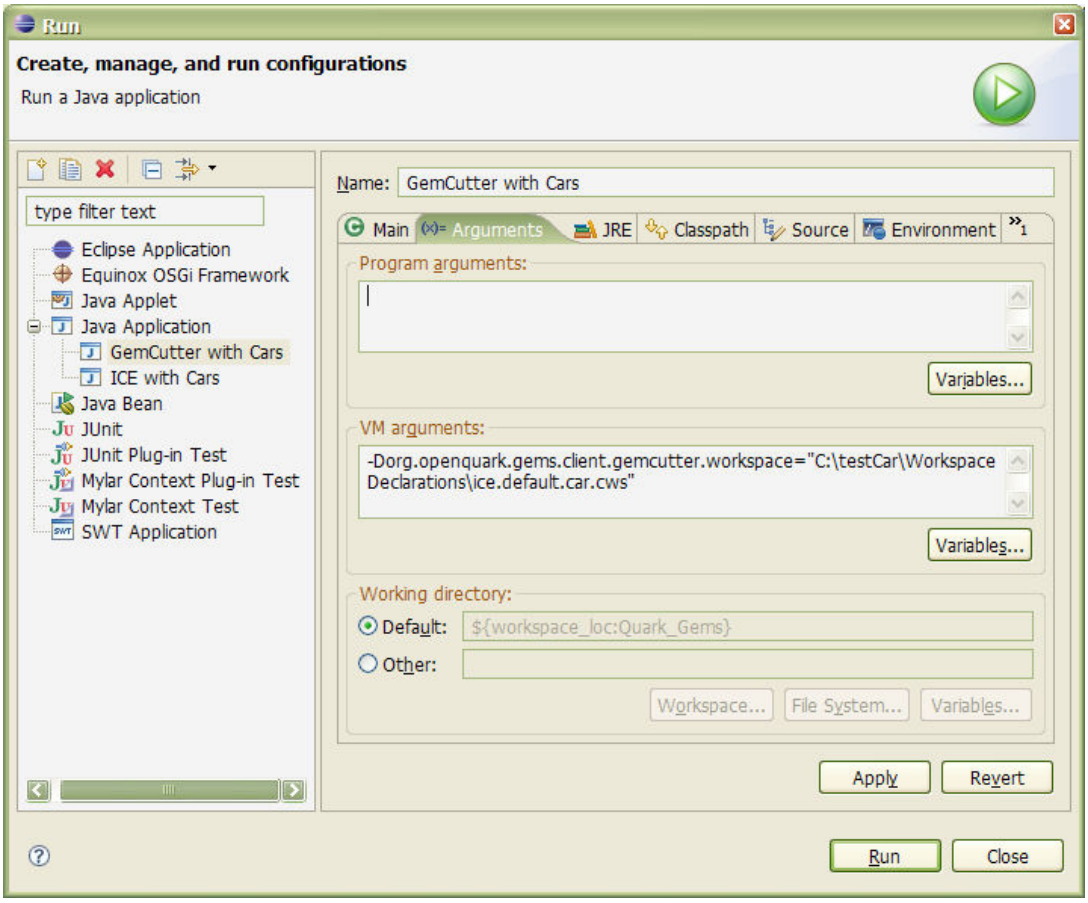
After this step, the run target's configuration is complete and can be launched.

4.1.2 GemCutter

The step to configure the classpath is the same as with ICE (see Section 4.1.1.1 above).

To configure GemCutter to use an alternate workspace declaration, we can use the system property "org.openquark.gems.client.gemcutter.workspace". The value of this property shall be the full path of one of the generated workspace declaration files, surrounded by double quotes.

For example:



4.2 Using Car-jar files

The CAL runtime will recognize Car-jar files added directly to the Java classpath.

For example, suppose one had generated some Car-jar files in ICE using:

```
:car -jar -d c:\testCar
```

Then, the Car-jar files in `c:\testCar` should be added to the Java classpath.

Also, one needs to inform the CAL program in question (e.g. ICE or GemCutter) to use one of the generated workspace declarations so that the CAL workspace will be initialized with the contents of the Cars.

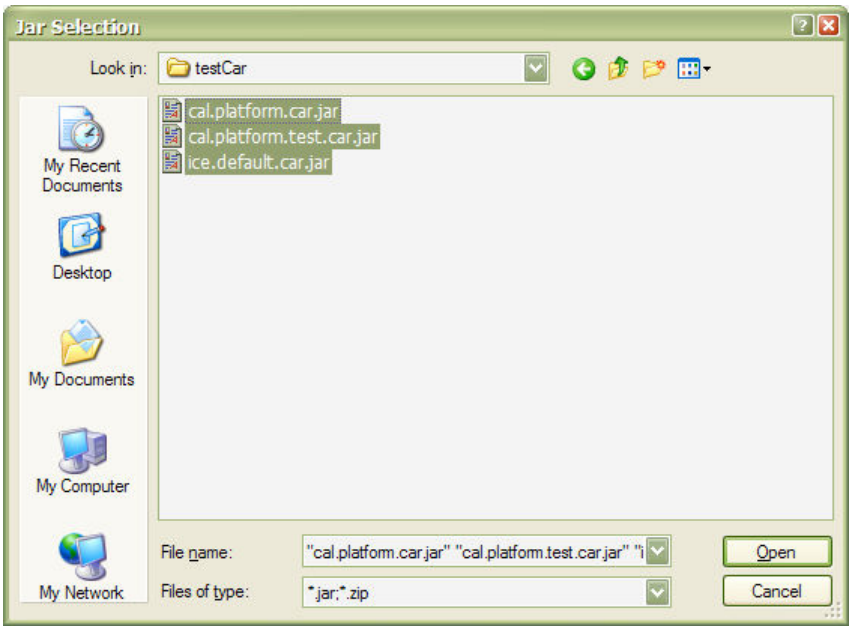
We will outline the entire process of setting up ICE and GemCutter in the Eclipse environment in the following sections.

4.2.1 Setup for ICE and GemCutter

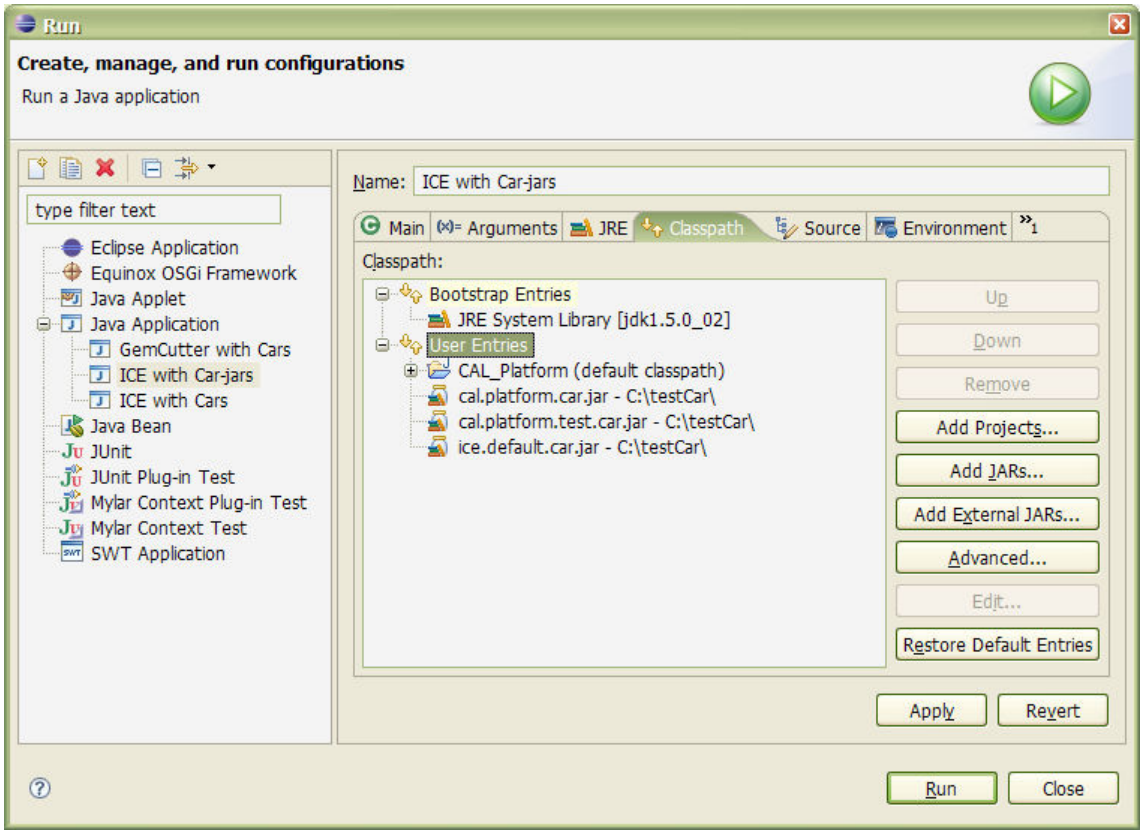
First, set up an Eclipse run target for either ICE or GemCutter – the steps are the same for both applications. For this example let us pick ICE.

4.2.1.1 Modifying the Java Classpath

To add the generated Car-jar files (which include the corresponding workspace declaration files) to the Java classpath, edit the properties of the run target and navigate to the Classpath tab. Once there, select the "User Entries" tree node in the Classpath panel, then click the "Add External JARs…" button. You can then multi-select all the generated Car-jar files, as shown:



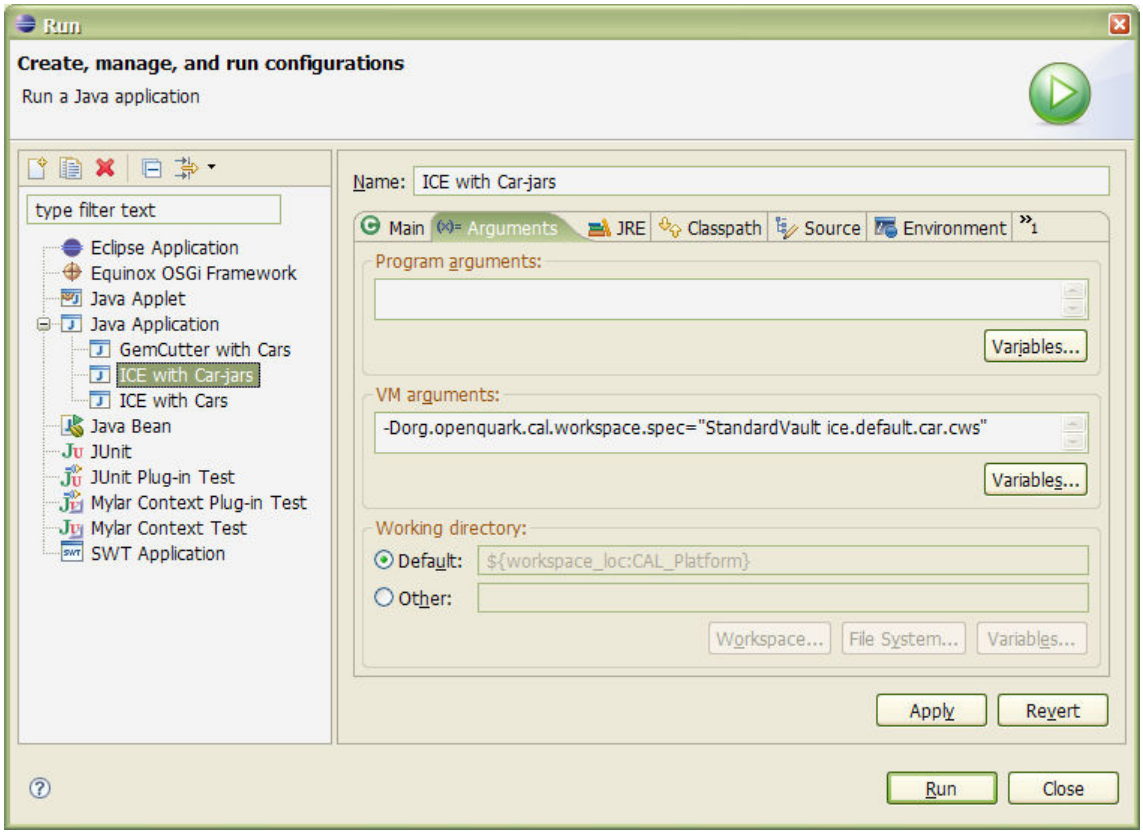
The final result should look like this:



4.2.1.2 Specifying an Alternate Workspace Declaration

Next, we need to tell ICE to use an alternate workspace declaration. This can be achieved by specifying the system property "org.openquark.cal.workspace.spec". The value of this property shall be a vault entry for the workspace declaration file, surrounded by double quotes.

For example:



After this step, the run target's configuration is complete and can be launched.

5 Deployment Scenarios for Cars

As designed, Car files work equally well within the confines of Eclipse as they do in a deployment scenario where the Quark Platform comes packaged in jar files.

There are a number of different ways we can use Car files in a deployment. One way is to simply provide a single Car containing all the required CAL modules and their resources.

We can also be a bit more sophisticated and provide multiple Carspec files (see Section 2.3) with the Car, so that the same Car can accommodate different use cases which require different sets of modules in their workspaces.

On the other hand, we can also package up the CAL modules into multiple Car files. For example, one approach would be to create a set of Cars where each Car corresponds to the modules listed in a particular workspace declaration file. (See Section 3.2)

One can use Car-jar files instead of regular Car files with any of the approaches listed above.

If the usage scenario for the deployment calls for the ability of the user to edit some of the CAL modules, these modules can be deployed separately outside the Cars. It is permissible for a CAL workspace to simultaneously refer to modules found in Cars, and to modules located in regular vaults (e.g. StandardVault).

