Johar Application Developers' Guide

0.1 Application Design Tasks

0.1.1 Identify Commands

Commands are the entry points with which the user is able to access the facilities of the application. If you are familiar with GUI applications, you may find it useful to think of commands as the menu items within each menu at the top of a typical application screen (for instance, "Open", "Close", "Save", and "Save As" in a typical "File" menu). However, you should keep in mind that not all interface interpreters will present commands to users as menu items.

It is often best to identify commands by thinking about what common tasks users will want to accomplish with your application, and what steps they will have to take in order to accomplish them. There are a lot of tradeoffs and design decisions to be made here. A typical tradeoff is between (a) having a small number of commands, each with many parameters, that allow the user to accomplish very specific goals with very few commands, and (b) having a larger number of commands, each with fewer parameters, that require users to go through more steps to accomplish their goals but allow them more flexibility.

0.1.2 Identify Parameters

When the user selects a command, they are really just selecting a single concept, word or phrase that describes in a general way what they want to do next. *Parameters* are the numeric, string and other data that goes along with a command and give more precise details about what the user wants to do. You will want to identify what parameters go with your commands.

Typical examples of parameters include:

- A page number to start printing at;
- A file to save data to;
- The width in centimetres of a graphic;
- The user's given name;
- Whether to sort data by name, date, or size; and
- Whether to sort in ascending or descending order.

Each parameter has a Johar *type*. The types of the above six example parameters would probably be `int`, `file`, `float`, `text`, `choice`, and `boolean`, respectively. The type of a parameter typically determines, in part, how an interface interpreter allows the user to specify the value for the parameter.

Each parameter can also have a default value. You might also need to decide whether the parameter is optional or not, and whether the user can give many values for the parameter (for instance, to give many filenames to process).

0.1.3 Identify Selection Data

For some parameters to your commands, your users will be selecting one or more items from a list of possible choices. You will want to identify this selection data. For instance, as mentioned above, the user

might want to select whether to sort data by name, date, or size; in this case, “by name”, “by date” and “by size” will be the choices. As another example, for an image library application, the user might want to edit information about an image they have entered into a database using the application; in this case, each image previously entered will be one of the choices.

The first question to ask about a kind of selection data is: For this kind of selection data, will the user have a fixed set of choices, that never changes except from one release of the application to the next? If this is the case, then it is easiest to code the selection data as a `choice` parameter. This will allow you to elicit the information from the user easily and still update the set of choices when you release a new version.

For example, say your application is a sound file converter which can convert to WAV or MP3 format. Later you might want to add the ability to convert to MP4 format, but for now those are the only two choices the user will have. You can code the selection as a `choice` parameter to some “Convert” command, giving “WAV” and “MP4” as the only two choices; later you can update the choice list in a new release.

On the other hand, if, for this selection data, the user will have a set of choices that may change during the course of a run, or from one run to the next, it is best to code the selection data as a Johar *table*. A table is a list of items that your application can update during the course of a run.

Tables can be further separated into *browsable* and *non-browsable* tables. Browsable tables are handled specially by interface interpreters; users are typically able to examine and select “rows” in the table and later specify what your application is supposed to do with the selected rows. This is a good choice for tables with a large number of entries, and where you imagine the user browsing through the entries in order to decide what to do with them. Non-browsable tables are not directly accessible to the user, but the user can choose from the values in the table to fill in parameter values. This is a good choice for tables with a small number of entries, or data that will probably not be the primary focus of the user’s browsing activities.

An example of a browsable table might be a table of images for the image library application. The user might have entered a large number of images, and may have several operations to apply to them, such as “delete image”, “add annotation”, and “email”. You might intend in your application for the user to spend much of their time looking at the image data and deciding which operation (if any) to apply to each image.

An example of a non-browsable table in the image library application might be a table of email addresses of the user’s contacts. While this is not fixed from run to run, and may even change within the course of one run, you might decide that for your application it is unlikely that the user will want to browse the data. In order to avoid cluttering the interface unnecessarily, you might therefore decide to make the table non-browsable, allowing the user to select email addresses from it only in the context of an “email” command.

0.1.4 Identify Quit Command

A small and easy but important task is to identify the command that the user will issue to quit the system. If you don’t add any command like this, then...

Hmm... maybe the IntI should automatically add one.

0.1.5 Refine Design

There are many refinements that you may end up making. Typical xxxxx...

Questions

Staged commands

Parameter value checking

Command groups

0.2 Implementation Overview

0.2.1 Setting Up the IDF and Application Engine

The Interface Description File (IDF) is what allows the user's interface interpreter to connect with the application. The application engine is the class that actually implements the operations that the user is trying to access in your application.

The initial setup of the IDF and application engine is simple, and starts with picking an application name. The easiest way is to pick a descriptive phrase (say "Image library"), capitalize each word and run them together to produce the name (for our example, `ImageLibrary`). Then create a file called `ImageLibrary.idf`, and put the following text into it:

```
Application = ImageLibrary
```

Then create a file called `ImageLibrary.java`, and put the following text into it:

```
class ImageLibrary {
}
```

You now have the skeleton of an IDF and application engine, which you will fill in in order to elaborate the commands, parameters and so on of your application.

Actually, the IDF can be named anything you want, and if you want the application engine class to be named something different from the application itself, then you can add a line to the IDF, e.g.

```
ApplicationEngine = "org.yoyodyne.ILAppEngine"
```

However, for clarity and simplicity, we recommend that the file names, the application name and the class name all correspond.

Johar IDF version number

0.2.2 Implementing Commands

Each command will correspond to an entry in the interface description file (IDF) and one method in the application engine class.

It is often easiest to name the command and the method by "camel-casing" the phrase for the command as you would want it to show up in a GUI menu. To camel-case a phrase, capitalize each word of the phrase, jam the words all together by eliminating spaces, and then make the first letter of the phrase lower-case.

For instance, commands that you would want to show up as "Open", "Save As", and "Use Settings As Default" in a menu would be named `open`, `saveAs`, and `useSettingsAsDefault`. If you are using this technique, the commands would be declared in the IDF as follows:

```
Command open = { ... }
Command saveAs = { ... }
Command useSettingsAsDefault = { ... }
```

where "..." is the full description of the command, as elaborated below. The declarations of the corresponding Java methods in the application engine class would be:

```
void open(Gem gem) { ... }
void saveAs(Gem gem) { ... }
void useSettingsAsDefault(Gem gem) { ... }
```

where "..." is the source code of the command method. Note that each of the methods must have a `void` return type, and must take exactly one parameter, of type `Gem`.

The phrase that corresponds to a command as presented to the user is called its *label*. If no explicit label is given for a command, a label is constructed from the command name by doing the inverse of the camel-casing operation (for instance, deriving "Use Settings As Default" from `useSettingsAsDefault`). However, in some cases it makes more sense to give an explicit label. For instance, for the command "Save As XML", if you call the command `saveAsXML`, it will get a default label of "Save As X M L"; the spaces in the phrase "X M L" appear because the de-camel-casing operation assumes that each of the letters is supposed to be a separate word. In this situation, it is better to explicitly define a label in the IDF, for instance as follows:

```
Command saveAsXML = {
  Label = "Save As XML"
  ...
}
```

It is also possible to name the Java method corresponding to the command differently, for instance by declaring the command in the IDF as follows:

```
Command saveAsXML = {
  Label = "Save As XML"
  CommandMethod = saveUsingXmlSaver
  ...
}
```

This would cause the interface interpreter to call a method `saveUsingXmlSaver` instead of `saveAsXML` in order to execute the command. However, it is recommended that you avoid this, in order to keep the IDF as simple as possible and the correspondence between IDF and application engine as straightforward as possible.

0.2.3 Implementing Numeric, Boolean and Text Parameters

Numeric parameters can be implemented by adding the appropriate named `Parameter` attribute to the IDF, for example:

```
Command print = {
  Parameter numberOfCopies = {
    Type = int
  }
  ...
}
```

In the corresponding command method in the application engine, you can access the value of such a parameter by using the `getIntParameter` method of the `Gem` class. Every command method takes a parameter of type `Gem`, and one of the uses of it is to get parameter values. For instance, the command method corresponding to the above `print` command might start like this:

```
void print(Gem gem) {
  int numberOfCopies = gem.getIntParameter("numberOfCopies");
  ...
}
```

Similarly, you specify a boolean parameter as type `boolean` and use method `Gem.getBooleanParameter` to access it; you specify a floating-point parameter as type `float` and use method `Gem.getFloatParameter` to access it; and you specify a text parameter as type `text` and use method `Gem.getStringParameter` to access it. (Note that the method for the `text` type is `getStringParameter`, not `getTextParameter`; more on this later.)

For integer and floating-point parameters, you can specify upper and lower bounds for the values to be entered, for example:

```
  Parameter numberOfCopies = {
    Type = int
    MinValue = 1
    MaxValue = 99
  }
```

One of the tasks of the user's interface interpreter is to check these bounds if they appear in the IDF, so the application engine can assume that any value it receives will be within the bounds.

Similarly, you can specify the maximum number of characters (`MaxNumberOfChars`) and the maximum number of lines (`MaxNumberOfLines`) that can appear in a text parameter. By default, the maximum number of characters is unlimited, but the maximum number of lines is 1. You can set the maximum number of lines to either an integer or the word `unlim`, meaning "unlimited". Again the interface interpreter is supposed to check these bounds, so you don't have to.

As with commands, parameters can be given a `Label` separate from the name of the parameter as it appears in the IDF. However, for simplicity and readability, this is not recommended. It's therefore important to think carefully about exactly what a parameter signifies before choosing a name for it.

0.2.4 Implementing Date and File Parameters

date

file

file constraints

0.2.5 Default Values, Optional Parameters and Parameter Repetitions

You can specify a default value for a parameter in the IDF with the `DefaultValue` attribute; for instance,

```
Command print = {
  Parameter numberOfCopies = {
    Type = int
    DefaultValue = 1
  }
  ...
}
```

The interface interpreter will fill in the default value if the user does not explicitly select a different value.

If the default value can change over the course of a run or from one run to the next, then what you probably want is to code a Java method that returns the default value. You can do this by specifying a `DefaultValueMethod`, for instance:

```
Parameter userName = {
  Type = string
  DefaultValueMethod = previouslyUsedUserName
}
```

Here, `previouslyUsedUserName` must correspond to a method in the Java application engine class declared as follows:

```
String previouslyUsedUserName(Gem gem) {...}
```

Note that this is declared identically to a command method, except that it has return type `String` rather than `void`. Default value methods for other types of parameters must return the appropriate type (`int`, `float`, `Date`, `File`, etc.).

If you identify a parameter as `Optional`, this means that the user doesn't have to give a value for it. More generally, you can give information in the IDF that allows the user to enter zero values, one single value, or more than one value for a parameter. We refer to these as "repetitions" of the parameter. The attribute `MinNumberOfReps` gives the minimum number of repetitions that the user can enter, and the attribute `MaxNumberOfReps` gives the maximum number. `MinNumberOfReps` can be any non-negative integer, and `MaxNumberOfReps` can be any positive integer or the word `unlim` (meaning "unlimited"). If these attributes are not given for a parameter, both default to 1, meaning that the parameter works as "normal"; that is, the user is expected to enter a single value (or let the default value be used). Specifying a parameter as `optional` is the same as saying that the minimum number of repetitions is 0 and the maximum is 1.

For instance, consider the following parameter declarations (not necessarily in the same command):

```
Parameter heading = {
  Type = string
  Optional
}
Parameter filesToProcess = {
  Type = file
  MaxNumberOfReps = unlim
}
Parameter filesToCompare = {
  Type = file
  MinNumberOfReps = 2
  MaxNumberOfReps = 2
}
```

The user can enter either 0 or 1 repetitions of the first parameter, and 1 or more of the second parameter, but must enter exactly 2 of the third parameter.

The command method can access information about the number of parameter repetitions by calling the `Gem` methods ... XXX In fact, the get... XXX methods are set up so that ..... XXX (exceptions)

Note the difference between a parameter with a default value and an optional parameter. If you declare a parameter as `Optional` (or if you declare it as having `MinNumberOfReps = 0`), then calling XXXX without checking the number of repetitions provided will result in an exception being thrown. This is the case whether or not you provide a default value for the parameter. If you provide a default value but the minimum number of repetitions is 1, then the interface interpreter will fill in the default value and you are guaranteed to be able to call XXXX at least once. In fact, the IntI is required to fill in the default value up to the minimum number of repetitions; of course, it also allows the user to select a different value if they want.

0.2.6 Implementing Value-Linked Parameters

Sometimes, giving a value for one parameter makes sense only if another parameter has a particular value. For instance, one might have a boolean parameter to a "print" command called `printToFile`; if the value of this parameter is `true`, then we would want the user to specify a file name to print to via another parameter, say `fileToPrintTo`, but otherwise it makes no sense for the user to specify a value for `fileToPrintTo`.

In such a situation.... XXXX

0.2.7 Implementing Selection Data

Choice parameter

Must turn tables visible/invisible to indicate no current data

0.2.8 Implementing Questions

0.2.9 Adding Help

0.2.10 Other Refinements

Prominence

Staged commands

Checking Parameter Values

Active and inactive commands

Quit after, if...

0.3 Recipe Book