Interfacing a Java graphical front-end to a Perfect Developer application

Introduction

The document Creating a Java application using Perfect Developer and the Java Development Kit describes how to build a Java console mode application using *Perfect Developer*. This document describes how to add a graphical front-end to a *Perfect* application compiled to Java. As an example, we will construct a graphical front-end to exercise the Dictionary example supplied with *Perfect Developer*. The complete application will create an empty dictionary and then allow the user to add words, remove words, and query whether words are in the dictionary. You can find all the files needed to build this application in the *Examples\Graphical* subdirectory of the *Perfect Developer* installation

Structure of a graphical *Perfect* application

Perfect Developer does not currently include a mechanism to easily construct graphical user interfaces (in part this is because only a few aspects of user interfaces are best treated formally). Therefore we recommend that an application with a graphical user interface be constructed in two parts:

- A back-end written in *Perfect* that handles all operations except interaction with the user;
- A graphical front-end written in Java (or some other language) which responds to user interaction by making appropriate calls to the *Perfect* back-end.

Typically, the back end will provide a single class for the graphical front-end to use. The front-end will create and maintain a single instance of this class.

Constructing the *Perfect* back end

Methods written in *Perfect* typically have preconditions and their parameters may have type constraints. If a method is only ever called from other *Perfect* methods, then *Perfect Developer* can verify that the preconditions are met. However, if a method is to be called from code written in another language, no such verification is possible. Therefore, we recommend that methods written in *Perfect* that are to be called from other languages should have no preconditions. This may require the construction of a wrapper class to act as a firewall between the *Perfect* back-end proper (whose methods have preconditions) and the user interface.

For our example, the back-end proper will be the *Dictionary* class as defined in the file *Dictionary.pd* to be found in the *Examples* subdirectory after *Perfect Developer* has been installed. We will need to invoke the constructor and the *check*, *add* and *remove* methods. The following preconditions and constraints apply:

- *check*, *add* and *remove* all take a parameter of type *Word*, which is a constrained version of **string**
- *add* has the precondition that the word to be added is not already in the dictionary
- *remove* has the precondition that the word to be removed is in the dictionary

We will use a new class *DictionaryWrapper* to act as a firewall between the user interface and the *Dictionary* class. Class *DictionaryWrapper* will do the following:

- Maintain a single *Dictionary* object

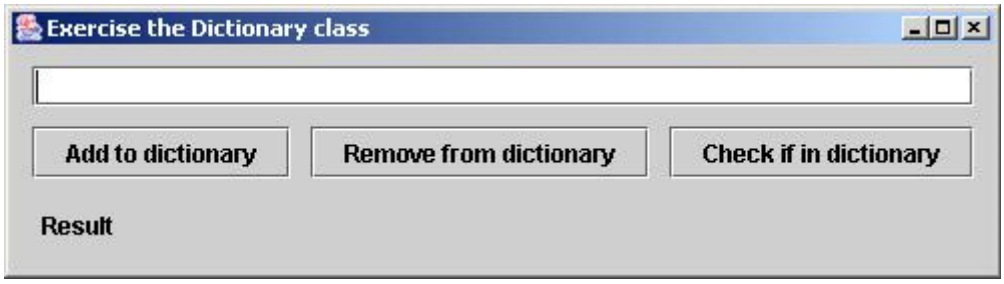
- Provide methods *check*, *add* and *remove* which front the corresponding methods of *Dictionary* but have no preconditions. Calls to these methods will only be passed on to the corresponding methods of *Dictionary* after checking the parameters. An error code shall be returned indicating whether the call was passed on or was refused.

For simplicity, we will have all three methods return error codes from a common set. We will define an enumeration class *DictionaryResult* to represent the set of return codes.

You can find our suggested specification of *DictionaryWrapper* and declaration of *DictionaryResult* in the file *DictionaryWrapper.pd* This file is successfully verified by *Perfect Developer* (in association with our original Dictionary.pd file), therefore it is certain that whatever calls the front-end makes to *DictionaryWrapper*, only valid calls will be made to the methods of the contained *Dictionary* object.

Constructing the Java front-end

For this example, we have constructed a simple graphical front-end using the Java Swing library. For more information on Swing, consult one of the many books available, or try this tutorial. Our graphical front-end looks like this:



Swing-based applications use a main class that implements interface *ActionListener*. Our application class will need a *DictionaryWrapper* to work on. So we declare our application class like this (see file *DictionaryAccess.java*):

```
public class DictionaryAccess implements ActionListener {
  DictionaryWrapper backend;
```

We also declare the variables needed for the graphical interface (in our case, a *JFrame*, a *JPanel*, a *JTextField*, a *JLabel* and three *JButton* objects).

The backend needs to be initialized, so in the constructor for *DictionaryAccess*, as well as initializing the graphical components, we include the following statement:

```
    backend = new DictionaryWrapper();
```

When one of the buttons is pressed, we want the word in the text box to be passed to the *add*, *remove* or *check* method of our *backend* object. This is done by implementing the *actionPerformed* method of interface *ActionListener* in class *DictionaryAccess*. However, before we look at the detail, we need to cover how *Perfect* identifiers, classes and types are translated into Java.

Converting between *Perfect* and Java identifiers

In Java, all user-defined names occupy a single namespace (e.g. you cannot declare a variable and a clsss with the same name). In *Perfect*, the names of constants, variables, functions etc. occupy a single namespace, but

class names live in a different namespace. Furthermore, the names of class templates with different number of template parameters occupy separate namespaces. So in *Perfect* the following declarations can happily coexist:

```
var foo: ...
class foo ^= ...
class foo of X ^= ...
```

In order to prevent clashes between declarations in the generated Java code, *Perfect Developer* performs name-mangling when translating identifiers, as follows:

- Class names that begin with a lower case letter are prefixed by **_n_**
- Template class names with 1, 2, ... template parameters are prefixed by **_n1_**, **_n2_**, ...
- Variable, constant, function etc. names that begin with an upper case letter are prefixed by **_n_**

So if you wish the generated Java names to match the *Perfect* names as far as possible, then in the *Perfect* source you should adopt the Java convention of starting class names with an uppercase letter and all other names with a lowercase letter.

Converting between *Perfect* and Java types

The *Perfect* types **int**, **char** and **byte** are represented directly by their Java equivalents; likewise, **bool** is represented by *boolean* and **real** is represented by *double*. So no translation is necessary when passing parameters of these types.

Abstract classes in *Perfect* that have no template parameters are translated directly to Java. This is why we were able to directly declare a Java variable of type *DictionaryWrapper* and invoke its constructor.

Enumerations are more complicated, because Java originally had no concept of an enumeration type. A *Perfect* enumeration class is translated into a Java class of the same name containing a single variable of type **int** called *value*. The actual enumeration names are declared as **public static final** members of type **int**. So for example, if in *Perfect* you declare:

```
class Color ^= enum red, green, blue end;
```

then in Java you can access the enumeration values as integers, e.g. "Colour.red". To actually construct a red *Color* object in Java (for example, to pass to a *Perfect* method) you would use `new Color(Color.red)`. The best way to test whether a *Color* object called *myColor* is red is to use the expression `myColor.value == Color.red`.

Passing objects that are instances of *Perfect* template classes is also complicated, because templates are not sufficiently expressive to support *Perfect* templates. This applies both to user-declared templates and to built-in templates like **set** and **seq**. The rules are:

- All instances of a given template are represented by a single class, which represents the template instantiated with each of its parameters replaced by type `from anything`.
- User-defined template names are translated to Java class names with the name-mangling already described (e.g. a user-defined `class Queue of X` translates to Java as `class _n1_Queue`).
- Built-in template names are translated into classes whose names begin with `_e`. So for example, `set of X` translates as `_eSet` and `seq of X` translates as `_eSeq`.
- When passing a parameter whose type is a template instance, the actual parameter is followed by one dummy parameter for each of the template arguments. This is required to resolve overloading. So a

variable *myList* of type **seq of real** would be passed like this:

```
(..., myList, (double) 0.0, ...)
```

If you have a Java variable of type *String* and you wish to pass it to a *Perfect* method expecting a **string**, you will need to convert it from the Java *String* representation to the *Perfect* **seq of char** representation (because **string** is equivalent to **seq of char**). Use the method *_eSystem._lString* to do this. So to pass a Java *String* variable *myJavaString* to a *Perfect* method, use:

```
(..., _eSystem._lString(myJavaString), (char) 0, ...)
```

You can convert the other way (from a *Perfect* **seq of char** to a Java *String*) using method *_eSystem._lJavaString*.

Armed with this information, you should be able to make sense of the *actionPerformed* method in the file *DictionaryAccess.java*.

To build and run the complete application, follow the instructions in Creating a Java application using Perfect Developer and the Java development Kit; but use *DictionaryAccess.java* in place of *Entry.java*, and use *DictionaryWrapper.pd* and *Dictionary.pd* in place of *Main.pd*.

Last updated March 2009