CLI2GUI Tool Proposal
Distributed Systems Services, EDS Bahrain

Ali Almossawi <ali.almossawi@eds.com>

1 Revision History

Version	Date	Changes
1.0	March 7, 2007	Initial version - DRAFT

2 Introduction

This document briefly discusses a tool that is to be written in the coming weeks in my free time barring no objection from my superior, Mr. Anwar Mirza. It is likely to prove useful for Gulf Air users who prefer interacting with a graphical user interface instead of a command line one. A couple of weeks ago, a user complained in passing about not liking to have to type things to get TD/Access to compress and decompress his files. The solution there was to write a little application[1] to graphically represent it. It was then realized that it might be interesting to generalize the idea and have a tool that can provide a graphical front-end for any command line application.

3 Motivation

Other than wanting to develop this tool because it seems like a fun project, the motivation for working on it is two-fold: to create an open-source Windows-based application that can easily provide front-ends to command line applications[2] and to get a first-hand sense of the added effort required to convert a specific solution that is tightly bound to some particular concept to a general solution that works with a much larger set of concepts.

4 Architecture

The proposed tool's architecture is shown in Figure 1.

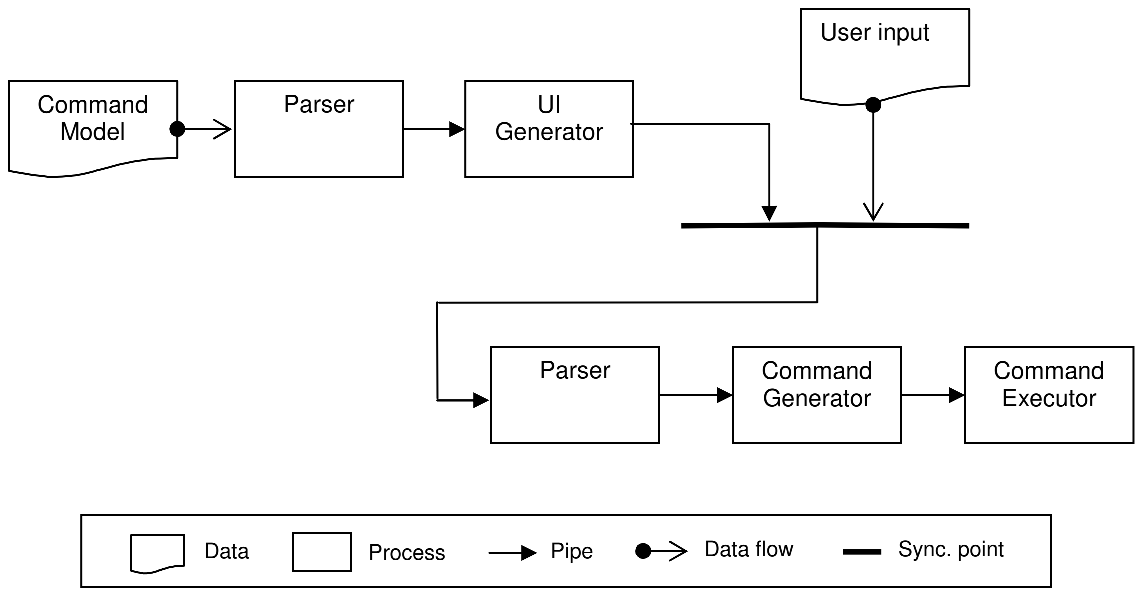

Figure 1: Runtime view of the tool represented using a pipe-and-filter architectural style

[1] The application can be obtained from the DSS team
[2] A *nix application called "Kaptain" is already available: http://kaptain.sourceforge.net

A *Command Model* is a representation of a command's syntax and semantics written using the notation described in section 5; it might be nice to extend it in the future to encompass other data such as heuristics. Once the *Command Model* is imported into the tool, it is parsed and validated then passed to the *UI Generator*, which dynamically builds the UI with appropriate input fields, labels and buttons. The tool then waits for the user to input all the data and proceed. Thereafter, the user-inputted data along with the *Command Model* are used to generate a command string, which is then simply executed.

5 CLI2GUI Notation

A straightforward notation is used to describe commands. There are three data types that the developer needs to worry about[3]:

- `arg_single`: A single argument

```
entity name:arg_single {
       value=string;
       label=string; //the label that the user sees
       required={0,1};
}
```

Code Listing 1: Syntax for defining an entity of type *arg_single*

- `arg_set`: A set of arguments containing zero or more elements

```
entity name:arg_set {
       label=string;
       required={0,1};
       enforce_order={0,1};
       delimiter=string; //delimiter used to separate set elements
}
```

Code Listing 2: Syntax for defining an entity of type *arg_set*

- `flag`: A flag prefixed with some user-defined character and followed by an *arg_set*

```
entity name:flag {
       value=string;
       label=string;
       required={0,1};
       prefix=string; //(e.g. "/" or "-")
}
```

Code Listing 3: Syntax for defining an entity of type *flag*

All one has to do is define **1)** the command that the model applies to, **2)** what the different parts, or entities, of the command string are, and finally **3)** the order in which they're all stringed together. Each data type has attributes, all of which must be filled out with the exception of the

[3] The code that defines an *arg_set*'s elements is automatically generated by the tool in response to user inputs via the GUI.

value attribute of the *arg_single* and *arg_set* data types, which are populated after the user input is received during the second parse.

The command must be defined on the first line as such:

```
command = string;
```

Defining the order in which arguments or flags appear in the command string is done in the following form:

```
command => argument or flag => argument or flag => ... ;
```

In the case of a flag, the *arg_set*, which can be used to append one or more options to it, is attached to the flag as such:

```
entity name_of_arg_set:arg_set ~name_of_flag { ... }
```

Now let's say the developer wants to write a *Command Model* for TD/Access' compress.exe, which has the following command string:

```
Compress src_path dest_path [options]
```

The entities here are the command, the source path, the destination path and the options. Something like the code in Code Listing 4 would be written:

```
command=compress;

entity src_path:arg_single {
       value="";
       label="Source path";
       required=1;
}

entity dest_path:arg_single {
       value="";
       label="Destination path";
       required=1;
}

entity options:arg_set {
       label="Options";
       required=0;
       enforce_order=0;
       delimeter="";
}

command => src_path => dest_path => options
```

Code Listing 4: Sample initial *Command Model* for TD/Access' compress.exe

The architectural diagram in section 4 doesn't show the logic involved in the user's interactions with the UI; one thing that must be noted is that the tool adds to the currently active *Command Model* during such an interaction in two main areas, hence the reason why a second parse is done:

1. The *value* attributes of blocks of type *arg_single* are populated with the values the user enters in their respective text fields.

2. The elements of entities of type *arg_set* are added in the following form:

```
entity auto_generated_name:arg_single ~name_of_arg_set { value=string; }
```

So for example, for the *options* block shown above, something like the following code would be generated based on the user's input, which tells the parser to add these three elements to the set with the name *options*:

```
entity options_0:arg_single ~options { value =“de3”; }
entity options_1:arg_single ~options { value =“ascii”; }
entity options_2:arg_single ~options { value =“crlf”; }
```

Code Listing 5: Sample code generated for compress.exe's *options* argument

Note that the rest of the attributes are automatically inherited from *options*.

Figure 2 shows a pictorial representation of a possible *Command Model* for compress.exe right before the second parse:

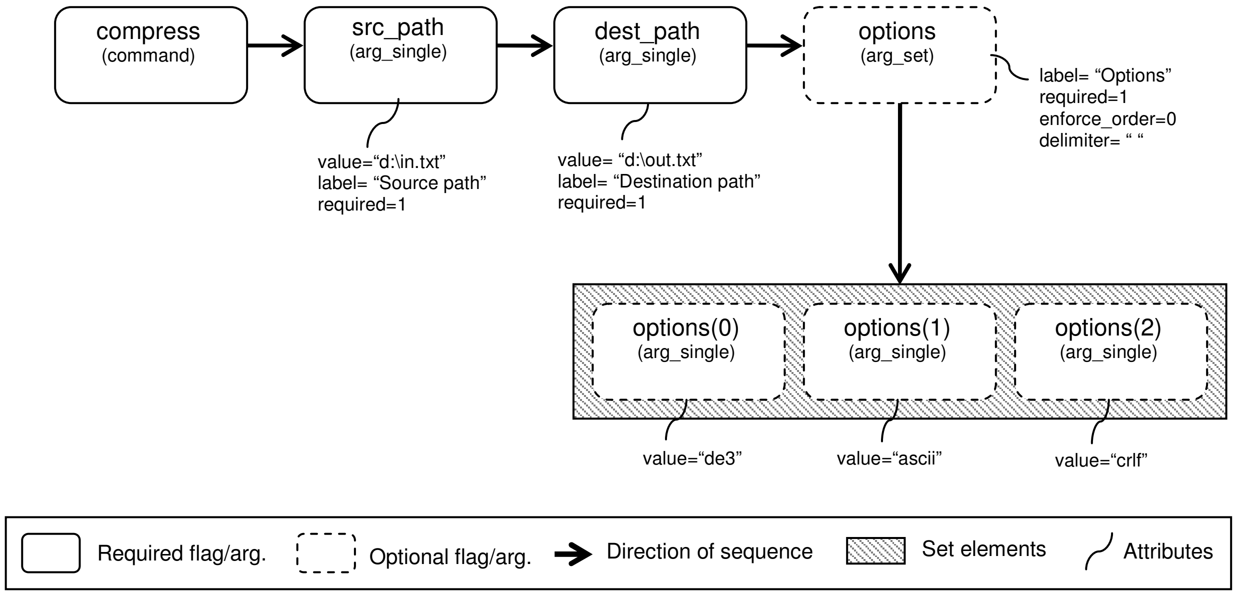

Figure 2: Pictorial representation of a possible compress.exe *Command Model* right before the second parse

6 Limitations

Confidence in the notation's ability to be accommodating to all or almost all Windows-based command-line applications, some of which may have unique styles, has yet to be tested and thus requires further investigation.