LibOpenCIF

April 8, 2014

User manual 1.0

1 Introduction

This section shows to the user the information needed to understand the idea and concept behind this library and the reasons to use it.

1.1 What is a CIF file?

A ***CIF*** file (*Caltech Intermediate Form*) is a special format designed to be used in ***IC*** (*Integrated Circuit*) design. Such format stores the physical layout of a design using basic figures (squares, metal stripes, polygons, circles). This format is used by various professional **CAD** (*Computer Aided Design*) tools, both open source and private, like ***Alliance VLSI*** (open source) and ***Tanner Tools*** (private).

One characteristic to keep in mind is that, in *IC* designs, one important aspect is precision. All the values managed (positions and sizes, for example), must have an incredible precision, since all the components must be precisely positioned and defined. To avoid problems relative to precision loss using floating point values, the values are defined as integers in the file. Also, the format doesn't limit the length of the values, so, the format of the values supported are defined by the applications that manipulate those values, not by the format itself (an application can support only 32 bit integers, but, others, can provide support for 64 bit values). That is the purpose of the AB fractions found in the definition start commands. Since there is still needed to use floating point values, the use of such special fractions help to translate the integer values to the exact floating values they are meant to be, and so, leaving the possible precision loss to the hands of the applications.

Such file format stores all the information of the design in plain text using commands. There are different kinds of commands that define and control different components of the design.

Box commands

Such commands define boxes (or rectangles) to be placed in the design. Such commands has three properties: position, size and an optional rotation vector, represented by a point.

Roundflash commands

Such commands define circles. The circles, however, normally are used as a piece component for the wire figures. The roundflash command has two properties: position and diameter.

Wire commands

Such commands define continuous stripes of material. The command is defined by a specific width and a list of points on which the stripe cross.

Polygon commands

Such commands represent arbitrary figures formed by a list of points.

Definition start commands

These commands represent the start of special groups of commands, named definitions. Those definitions can represent anything, from a small cell of a design, to a whole design. Such commands specify an identification ID (a number), and, also, an optional AB fraction that can be used to transform all the values of the commands contained in the definition itself, from integers to floating point values.

Layer commands

The layer commands specify the material used by the commands that follow them.

Definition end commands

These commands help to set the end boundaries of the definitions.

Definition delete commands

The definitions already specified can be used as needed, all the times needed. If, for some reason, is needed to use an identification ID already used, is possible to use this command to delete from memory a used definition.

Call commands

The format itself defines that nothing in the file does something for the design until there is a call to a definition (that means that there can be things defined but never used). The call commands specifies the usage of a specific definition. The calls support the specification of some transformations to be applied to such definitions, like rotation, displacement and mirroring. Also, the calls can be put inside of the definitions, so, a big cell in a design can be split into smaller pieces, called as needed.

Extension commands

The extension commands are used to expand the possibilities of a *CIF* file. Such commands can be anything, and are application dependent. So, if a toolset A defines an expansion command with some special format to display text within the design, such command will likely only work in that toolset A, but not in the toolset B, that, maybe, has the same functionality, but the internal command used to do the same job is incompatible.

There are some basic extension commands that are widely used across various *CAD* tools, like the cell name command ("9 name;"). This library doesn't validate the extension commands, since there is none standard regulation for them. So, they are used as regular comments, and the validation of their contents are left to the user.

End command

The end command defines the end of the file.

1.2 What is LibOpenCIF?

LibOpenCIF is an open source library intended to aid the programmers to be able to load a *CIF* file into memory. The library provides basic validation of the contents. All the validations are performed using a special designed finite state machine.

The *CIF* file format, even defining a syntax of how to write a correct file, an input file can be hard to read, since the white spaces can be a lot of things, from real white spaces (spaces, tabs, enter) to lower case chars (all the lower case letters, from 'a' to 'z'). That means that there is possible to found a command like this one:

```
‘‘B100xjnjzncjjajnja,,as,c,cac,„,ac,csc,100,,ca,ca,cas,s,„,gh,h50,-50,,as-1,-1;’’
```

That, surprisingly, is a completely valid command. So, all the applications that need to load a CIF file, needs a way to load the contents and extract the real information from the commands. LibOpenCIF provides the ways to easily validate and load the contents of the input files. In this example, the library will provide the user with two basic things: A cleaned and easy to manage string, containing the same command and a class instance that provides direct and instant access to the values of the command. So, as an example, the previous command can be loaded with the library in the following form:

```
‘‘B 100 100 50 -50 -1 -1 ;’’
```

So, the library gives to the user two different mechanisms to work with the information of a *CIF* file.

1.3 Why use LibOpenCIF?

LibOpenCIF can be used by everyone. From small programs, to big ones. These are some of the advantages of using LibOpenCIF on your system:

- Pure and standard C++ code: The library is self contained. That means that, from the compilation process, to the usage, there is no dependence in any other library or special tool. All of the capabilities of the library are provided using only C++ code and none system-call.

- Input file validations: The contents of a CIF file must be validated to ensure that the contents are complete and correct commands. Such validations are performed by the library using a special finite state machine, designed to validate and support various command formats found in use in professional tools (like layer names with more than 4 characters, digits and underscores). All of this ensures that the load process is performed according to the CIF specifications, but being compatible with industrial applications.
- Contents formatted for the user: After validating the input file contents, the library perform a buffering of the commands. This means that all the commands found in the input file can be accessed as a vector of strings. Also, the commands are not just copied from the input file. The string commands buffered are cleaned. That means, that all the separation chars are replaced by single white spaces. This formatting helps to have all the components of the command separated by a white space, allowing the user to use, for example, the strings as input streams.
- Contents ready to use: The library also offers to the user a ready-to-use vector of pointers. Such pointers, of a basic class, are instances of specialized classes that represent the commands of the file.
- Output and input operators: The classes used by the library provide the mechanisms to let the user send and read the commands from wherever he wants[1].

[1] Read the section **??** for more about such topics.

2 Using LibOpenCIF

The following sections shows the user how to use the library, from installing and uninstalling, to the common usage of the loaded results from the `CIF` files.

2.1 Installing

The installation is very straight forward. First, you need to get the library package, a compressed file containing all the sources and other specific files needed to install it. You need a file named *libopencif-X.Y.tar.gz*, where X and Y goes for the mayor and minor release number of the package.

Once you get the package, extract it in a new folder with full permissions of the actual user (like a folder in your home folder or any extra partition with write permissions, but, preferably not in a removable device like a USB stick or an external hard disk).

After extracting the package, open a terminal and change it's active working directory to the folder where the contents of the package are (so, that when you execute the `ls` command, you can see listed the contents).

On the extracted folder, with a terminal window opened, run the following commands, one by one, as a normal user (not as root nor using *sudo*, except for the last one). Don't forget to check the output of the commands to find any problem:

```
$ ./configure
$ make
$ sudo make install
```

Remember, the "$" simbol is not part of the command. After running the commands, the library should be correctly installed in your system.

2.2 Uninstalling

If the library needs to be uninstalled, you can do it using the library package itself. To uninstall, get the package file, extract it's contents (as done in the installation procedure) and run these commands:

```
$ ./configure
$ sudo make uninstall
```

The following sections demonstrate the usage of the library. All of the relevant components of the library, from a user point of view, are covered. If you need more information, refer to the Technical Manual.

2.3 Compiling a program with the library

The first important point to learn how to do is the compilation of a program using the library. The compilation is very simple.

Let's assume you have only one source file, named `program.cc`, a C++ program source code. So, the compilation can be done with this command:

```
$ g++ program.cc -o binary_name -lopencif
```

Such command calls the compiler, named `g++`, followed by the source program file `program.cc`. After that, the `-o binary_name` option specifies the name of the final binary file. Finally, there can be found the option that will add the library itself to the compilation process: `-lopencif`.

After running such command, the program will be compiled and ready to run. Please, refer to the documentation of your compiler or IDE to be able to add the needed option (`-lopencif`) to the compilation process.

2.4 Opening a CIF file

After learning how to compile your program, the first important task to perform, when using the library, is how to open a *CIF* file. For this example, we will assume that your application source file is named `main.cc` and your design file is named `design.cif`. Also, there is the assumption that both files are located in the same folder.

First, lets introduce the basic C++ program code that will help to demonstrate the usage of the library on the algorithm 1.

Algorithm 1 Initial test program source.

```
1 # include <iostream>
2
3 using namespace std;
4
5 int main ()
6 {
7    cout << "Test program" << endl;
8    return ( 0 );
9 }
```

Such code is a standard C++ program. The program, by itself, does nothing but print a simple message on the terminal. To use the library, the first step is to include the library itself. To do it, add this include instruction:

```
# include <opencif/opencif>
```

After including such line, all the components of the library are now available to use in the program. All the components of the library can be found within a namespace named `OpenCIF`. To open our design, you need to create an instance of the class `File`.

```
OpenCIF::File design;
```

The instance can also be created using dynamic memory allocation (using `new` and pointers).

After such instance is ready, we need to set the actual path to the file to open. The path can be relative or full. To set the path, the `File` class have a member function named `setPath`, which takes as argument a C++ `string` class instance or a direct string.

```
design.setPath ( "design.cif" );
```

After that, the instance is ready to be used. The next step is loading the file.

2.5 Loading the whole file

After setting the file path, we can now order the `File` class instance to load the contents of the file.

The library uses, by default, an automated process of loading. Such process performs the following tasks in order:

1. Open the file.
2. Using a finite state machine, start validating the contents of the file, char by char. At the same time, the library is buffering the commands found and validated in a string form.
3. After loading the file into memory, the library performs a cleaning process to the commands loaded, replacing the separation chars with single white spaces.
4. Finally, the library converts the commands buffered into class instances.

Those 4 steps are performed automatically with a single member function call, named `loadFile`. The `loadFile` member function calls the needed member functions of the `File` class to perform the previous four tasks[2]. Of course, the tasks one and two can fail by some reasons. So, the library has implemented a special value type that is returned when calling the `loadFile` member function and helps the user to know if there is any kind of problem with the process.

The possible values of the return value are constants that can be found within the `File` class. The type of such constants is `LoadStatus`. So, to ask the `File` instance to load the file and store the result of the operation, you need to create a variable of type `LoadStatus` and call the `loadFile` member function from your instance:

[2]To learn more about a manual load process, please, refer to the section 2.8 on this document.

```
OpenCIF::File::LoadStatus end_status;
end_status = design.loadFile ();
```

The possible values of the `LoadStatus` type are four constants found in the `File` class definition. The following piece of code checks the result of the operation. Take in mind that this is only an example, so, you can use a different approach to perform a validation of the result:

```
switch ( end_status )
{
   case OpenCIF::File::AllOk:
      cout << "None error detected" << endl;
      break;
   case OpenCIF::File::CantOpenInputFile:
      cout << "Error opening file" << endl;
      break;
   case OpenCIF::File::IncompleteInputFile:
      cout << "Incomplete file, no End Command found" << endl;
      break;
   case OpenCIF::File::IncorrectInputFile:
      cout << "Incorrect file" << endl;
      break;
}
```

These are the four possible values of the `LoadStatus` type:

AllOk This value is returned if all the operations where performed correctly. That means: The file can be opened, the contents of the file are correct and the end command was found.

CantOpenInputFile This value is returned if there was a problem opening the input file. Such problem can be related to the file path (the path can be incorrect, the file was removed before opening, etc).

IncompleteInputFile This value is returned if the input file is *correct*, but there was not found the end command. Even when this command is returned, all the validated and cleaned commands (in string form) are still stored and accessible for the user. If the user needs, he can manually call the member function related to transform the loaded string commands into class instances. The user must be aware that the contents, being incomplete, can be not useful at all.

IncorrectInputFile This value is returned if the validation process has reached an unexpected character in the file. In this case, in a similar way that the previous return value, the validated commands are still buffered and accessible to the user. Of course, there is a way to force the complete loading of a file, even when incorrect commands can be found. Such option will be explained in the section 2.8 found in this document.

2.6 Using the strings loaded

If you have a file already loaded, that means that you have received either an `AllOk` value (a complete loaded file) or any other value (a partial load, at best) from the `loadFile` member function, then you can make use of the loaded commands as you need. The commands are stored by the library in a C++ `vector` class that stores `string` class instances. Each `string` instance contains a full and valid *CIF* command. For every command in the *CIF* file, there is a `string` counterpart on the vector, in the same order.

To gather access to such commands, you must use the member function `getRawCommands`.

```
vector< string > commands;
commands = design.getRawCommands ();
```

After getting the commands, you can use them as needed. The format of the commands is very clear and easy to understand[3]. All the commands have it's components separated by single white spaces, so, all the separation characters are replaced. The next are two valid commands in CIF format (the first one is an exaggeration, but yet correct, the second one is as can be found in some professional tools):

[3]Refer to the section 3.2, or in the Technical Manual, to learn more about the cleaning process of the commands.

```
Pthis,is,an,example100,-100,,,200,100more,text,100,0,,,;
P100,100 200,100 100,0;
```

So, the library transform such command to this:

```
P 100 -100 200 200 100 0 ;
```

All the components (included the identification character, the 'P' char and the semicolon) are separated with single white spaces. This format is intended to let the user use any process he wants to extract the information. The white spaces are used since they can be easily identified and can be used in common algorithms for splitting strings. Also, this command format is useful if the user wants to read the values of the command using a C++ input stream, like a `istringstream` class instance.

2.7 Using the instances created

If the file is correctly loaded or the partial loaded contents are manually ordered to be converted into instances (see section 2.8), the user can have access to a C++ `vector` class instance that stores class pointers. Those pointers are of the `Command` class, a special common base class used in most of the hierarchy of the classes defined in the library, so, using polymorphism, the user can ask, pointer by pointer, which command type is (a Box, Wire, Polygon, etc) and cast the pointer as needed (only the member function that returns the type of the command is polymorphic). The next example code shows how to get access to the vector of pointers from the `File` class instance using the `getCommands` member function.

```
vector< OpenCIF::Command* > commands;
commands = design.getCommands ();
```

After getting the commands, you can use the polymorphic member function `type` to get a constant value that represent the command type. The next example code, shows a simple process where the commands are already stored in a vector named `commands`. The code uses a `for` cycle to check all the commands stored and ask them their types using a switch structure. In every case, the pointer is casted to a derived class pointer and then printed to the console.

```
vector< OpenCIF::Command* > commands;
commands = design.getCommands ();
for ( int i = 0; i < commands.size (); i++ )
{
   OpenCIF::BoxCommand* box;
   /*... More class pointers of every type ...*/
   OpenCIF::Command* cmd;
   cmd = commands[ i ];
   switch ( cmd->type () )
   {
      case OpenCIF::Command::Box:
         box = dynamic_cast< OpenCIF::BoxCommand* > ( cmd );
         cout << box << endl;
         break;
      /*... More Switch cases ...*/
      default:
         break;
   }
}
```

In this little example, a `for` cycle traverse over all the commands loaded from the file. For every command, it's pointer is copied into a `Command` class pointer (only to reduce the name from "`commands[ i ]`" to "`cmd`"). After that, a switch is used to check the command type.

To access the command type, there is possible to use the polymorphic member function `type`. Such call will return a constant value of an enumeration type. Since, in C++, if a `switch` structure is used to check the value of a `enumeration` type, all the possible values of such `enumeration` should be used. That is the reason to use the

empty `default` statement, to avoid the need to include unnecessary cases (there are various class types defined in the library that are not intended to be used by the user).

In this case, there only appears to be used one value of such enumeration (see section 2.8.6 to learn more about the classes intended to be used by a user), the `Box` value (all the values that represent the type of a command are stored in the `Command` class within the `OpenCIF` namespace).

In this case, the `cmd` pointer is dynamically casted into a `BoxCommand` class pointer. With such casted pointer, the command can be printed[4].

Sadly, the member functions of the classes are not available using the base pointer beyond the `type` member function (they are not polymorphic). That means that there is needed to perform the casting to derived pointer in order to use the needed member functions.

2.8 Manual loading process

There is possible that you need or want to load a CIF file until some specific step and/or want to force the load of the file even when facing errors in the contents. The LibOpenCIF library provides such facilities to the user.

2.8.1 Opening the input file

The first process to review is the manual loading. To manually load a file, the first step, as in the automatic method, is to set the file path using the `setPath` member function:

```
OpenCIF::File design;
design.setPath ( "design.cif" );
```

After that, the first manual operation to do is opening the input file. To manually open the file, use the `openFile` member function.

```
design.openFile ();
```

Such call can return two different results:

AllOk The file was opened correctly.

CantOpenInputFile The file can't be opened. In this case, the user must perform some kind of validation of the file path, from checking that it is correct, to validate the existence and permissions of the file itself.

The user must be aware that this step is needed for the rest of the steps.

2.8.2 Validating/buffering file

The next step, once the input file is opened, is to validate the file. The user must remember that, while validating, the library performs a buffering of the commands found, storing them as strings. This process is performed by the call to the `validateSyntax` member function.

```
design.validateSyntax ();
```

Such call can return three possible values:

AllOk The input file is correct and complete.

IncompleteInputFile The input file is correct, but there was not found the End command. This may result in an incomplete and useless design file[5].

IncorrectInputFile The input file has errors. Normally, after detecting an unexpected character, the validation/bufferin process stops.

In all the three cases, the commands buffered and validated are accessible through the call to the `getRawCommands` member function. Such call will return a vector of strings. Such strings are the commands found in the input file, but they are still not cleaned, so, you will find them exactly as you can see them in the file. If you need exactly that, this is the point when you can retrieve the commands.

[4]The commands are designed to be printable from 3 states: base pointer (as `Command` class pointers), class pointers (as in this example) and as regular class instances (not pointers).

[5]There must be pointed that some professional applications support such lack of command, but that doesn't mean that a valid CIF file must not have an End Command.

2.8.3 Cleaning the commands

After loading the input file, a call to the `cleanCommands` member function is recommended, since it performs the cleaning of all the commands loaded (no matter if the file was correct nor complete).

```
design.cleanCommands ();
```

All the buffered commands are replaced with their cleaned counterparts (to reduce memory usage). If there is needed to keep a copy of the original commands of the file, is recommended to perform a call to the `getRawCommands` member function and storing them in a string vector before calling the member function to clean the commands.

This member function doesn't return any value and can't fail (since it only works with the commands loaded, if there is none, will do nothing).

2.8.4 Converting the commands into instances

The last step of the load process. If there are commands loaded and cleaned, the member function `convertCommands` can be called to convert them into class instances.

```
design.convertCommands ();
```

After calling it, the command class pointers can be accessed using the `getCommands` member function.

The `convertCommands` member function doesn't return anything, but the user must be aware that this member function is the most prone to errors, since it's design and work flow depends on the string commands stored. The commands must be already cleaned. ***If the commands are not cleaned, the call to this member function can lead to logical or even critical errors (program crashes)***. Remember: The member function assumes that the commands are clean, so it can work a little bit more faster and prevent bottlenecks.

If the user is totally sure about the nature of the input files and their formats, the cleaning operation can be skipped to increase the overall speed of the load process.

2.8.5 Forcing load of file

There is possible to force the process of validation of a file, so it skips the incorrect commands and load as much as possible of a file. The user must be aware that doing this may result in an incorrect design that may not be useful at all.

The process is simple, and can be used when automatically or manually loading a file.

When using the automatic process, calling the `loadFile` member function of the `File` class, you can pass a constant value to it to specify what to do when encountering an error. By default, a call to such member function in this way:

```
design.loadFile ();
```

Will automatically pass the constant value `StopOnError` that specify the behavior of the process when confronting an incorrect or incomplete file. Such value is located within the `File` class. So, to explicitly order the automatic load process to stop when facing errors, you can use this call:

```
design.loadFile ( OpenCIF::File::StopOnError );
```

Also, you can force the automatic process to continue loading and processing the commands found even when facing an incomplete or incorrect command. To do it, you can use the constant value `ContinueOnError` that is also found within the `File` class. To use it when automatically loading a file, you can use a call like this one:

```
design.loadFile ( OpenCIF::File::ContinueOnError );
```

Doing this, the process will ignore the fact that a file is incomplete and will try to skip incorrect commands and read as many as possible[6].

Even using this two constants, the `loadFile` member function will still return the correspondent value after loading, either `AllOk`, `IncompleteInputFile` or `IncorrectInputFile`.

Those constants (`StopOnError` and `ContinueOnError`) can also be used when manually loading a file. From all the member functions that can be used, the only one that accept such values is the `validateSyntax` member

[6]There must be noted that the error CantOpenInputFile will still stop all the process, since is a critical error that can't be ignored.

function. All the other member functions don't need such control, since they operate on previous results or are truly critical processes (like the `openFile` member function), so, the `validateSyntax` member function, by default, uses the `StopOnError` constant, but you can use the specific constant that you need. You can specify an instruction like this one:

```
design.validateSyntax ( OpenCIF::File::StopOnError );
```

So, the call to such member function will behave normally. After finding an unexpected input char, the process will stop. But, you can specify other constant to force the validation of the file, even when facing errors in the input file:

```
design.validateSyntax ( OpenCIF::File::ContinueOnError );
```

In this case, the member function will continue and skip as many errors as possible.

2.8.6 Classes intended to be used by the user

When loading a CIF file, you can use the `OpenCIF` library to automatically convert the CIF string commands into OpenCIF class instances. As explained before, you can do it when loading a file in automatic mode, or when manually loading it's contents.

Remember that all of this classes can be found within the `OpenCIF` namespace. The most important and useful classes for the user are the next ones.

Command

This is the base for most of the class hierarchy in the library. The next list shows all the relevant member functions, constructors and destructors that the user might need when using this class.

- `Command`: Constructor of the class. Takes no arguments and initialize the instance type as a `Command`.
- `type`: Public member function that is polymorphic. Takes no arguments and returns a value of type `CommandType`. In the case of this class, the value returned is `PlainCommand`.

The class has, also, defined two operator functions that let the class be written and read to and from an input and output stream, respectively[7]. Those operators expect a class pointer, not a normal instance. This is due that this class is intended to be used only to store other classes (base-class-pointer to derived-class mechanism). These two operators are not polymorphic for themselves, but they call internal private member functions that are polymorphic. So, sending a class instance using a Command pointer to an output stream, will call the needed function to convert the instance into a string and write it to the stream, or, if extracting from a stream, the operator will call the relevant polymorphic function to read an instance.

Point

This class represents a point in the file and is used in various classes in the library. The points can have any value in their X,Y coordinates, negative and positive. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `Point`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance with values 0 in X and Y (the origin of a design). The other constructor takes as argument two `long int` integers representing X and Y, in that order.
- `set`: Public member function that helps to set the coordinates of the point in a single call. Takes as argument two `long int` integers, representing X and Y, in that order. Has no return value.
- `setX`: Public member function that sets the X component of the point. Takes as argument a `long int` integer value. Has no return value.

[7] The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

- `setY`: Public member function that sets the Y component of the point. Takes as argument a `long int` integer value. Has no return value.
- `getX`: Public member function that returns the X component of the point. Returns a `long int` integer value. Takes no arguments.
- `getY`: Public member function that returns the Y component of the point. Returns a `long int` integer value. Takes no arguments.

The `Point` class has, also, defined two operator functions that let the instances be written and read to and from an input and output stream, respectively[8]. Those operators expect a normal instance, not a pointer. The next code lines represent the calls presented.

```
// Normal instance, default constructor (point initialized to 0,0)
OpenCIF::Point point;

// Normal instance, overloaded constructor (point initialized to 100,100)
OpenCIF::Point point ( 100 , 100 );

// Manually setting the point values
point.set ( -100 , 200 );
point.setX ( -100 );
point.setY ( 200 );

// Manually getting the point values
long int x , y;
x = point.getX ();
y = point.getY ();

// Reading a point from an input file
ifstream file;
/*... Opens file ...*/
file >> point;

// Writing a point to an output file
ofstream file;
/*... Opens file ...*/
file << point;
```

Size

This class represents the size of some figures in a *CIF* file and is used in some classes in the library. The size has two components: width and height. As can be expected, both values can't be negative. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `Size`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance with a size of 1,1. The other constructor takes as argument two `unsigned long int` integers representing width and height, in that order.
- `set`: Public member function that helps to set the component size in a single call. Takes as argument two `unsigned long int` integers, representing width and height, in that order. Has no return value.
- `setWidth`: Public member function that sets the width component of the size. Takes as argument an `unsigned long int` integer value. Has no return value.
- `setHeight`: Public member function that sets the height component of the size. Takes as argument an `unsigned long int` integer value. Has no return value.

[8]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

- `getWidth`: Public member function that returns the width component of the size. Returns an `unsigned long int` integer value. Takes no arguments.
- `getHeight`: Public member function that returns the height component of the point. Returns an `unsigned long int` integer value. Takes no arguments.

The class has defined two operator functions that let the class be written and read to and from an input and output stream, respectively[9]. Those operators expect a normal instance, not a pointer. The next code lines represent the calls presented.

```
// Normal instance, default constructor (size initialized to 1,1)
OpenCIF::Size size;

// Normal instance, overloaded constructor
OpenCIF::Size size ( 100 , 100 );

// Manually setting the size values
size.set ( 100 , 200 );
size.setWidth ( 100 );
size.setHeight ( 200 );

// Manually getting the size values
unsigned long int x , y;
x = size.getWidth ();
y = size.getHeight ();

// Reading a size from an input file
ifstream file;
/*... Opens file ...*/
file >> size;

// Writing a size to an output file
ofstream file;
/*... Opens file ...*/
file << size;
```

Fraction

This class represents an AB fraction. The AB fractions are used within the definition start commands in the file. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `Fraction`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance with a neutral value of $\frac{1}{1}$ (neutral since, when applied, does not affect any value). The other constructor takes as argument two `unsigned long int` integers representing the numerator and denominator, in that order.
- `set`: Public member function that helps to set the fraction value. Takes as argument two `unsigned long int` integers, representing numerator and denominator, in that order. Has no return value.
- `setNumerator`: Public member function that sets the numerator component of the fraction. Takes as argument an `unsigned long int` integer value. Has no return value.
- `setDenominator`: Public member function that sets the denominator component of the fraction. Takes as argument an `unsigned long int` integer value. Has no return value.
- `getNumerator`: Public member function that returns the numerator component of the fraction. Returns an `unsigned long int` integer value. Take no arguments.

[9]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

- `getDenominator`: Public member function that returns the denominator component of the fraction. Returns an `unsigned long int` integer value. Takes no arguments.

The class has defined two operator functions that let the class be written and read to and from an input and output stream, respectively[10]. Those operators expect a normal instance, not a pointer. The next code lines represent the calls presented.

```
// Normal instance, default constructor (size initialized to 1/1)
OpenCIF::Fraction fraction;

// Normal instance, overloaded constructor (initialized as one half)
OpenCIF::Fraction fraction ( 1 , 2 );

// Manually setting the fraction values (setting values as one half)
fraction.set ( 1 , 2 );
fraction.setNumerator ( 1 );
fraction.setDenominator ( 2 );

// Manually getting the fraction values
unsigned long int n , d;
n = fraction.getNumerator ();
d = fraction.getDenominator ();

// Reading a fraction from an input file
ifstream file;
/*... Opens file ...*/
file >> fraction;

// Writing a fraction to an output file
ofstream file;
/*... Opens file ...*/
file << fraction;
```

Transformation

This class represents a transformation. The transformation can be found in the call commands. The class doesn't have any special member function to represent the mirroring, since there is only needed to know in witch direction the transformation is going, and that is configured with the transformation type. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `Transformation`: Constructor of the class. The constructor, by default, creates the transformation as a neutral displacement (a 0,0 displacement).
- `setType`: Public member function that helps to set the transformation type (a single transformation instance can be configured to be either a displacement, rotation or mirroring). Takes as argument a constant value from an enumeration defined within the Transformation class. The possible values are:
  - `Displacement`
  - `Rotation`
  - `HorizontalMirroring`
  - `VerticalMirroring`
- `getType`: Public member function that returns the type of the Transformation. The possible values to be returned are:
  - `Displacement`

[10]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

  - `Rotation`
  - `HorizontalMirroring`
  - `VerticalMirroring`

- `setRotation`: If the Transformation instance is configured as a `Rotation`, this public member function sets the `Point` class instance that represents the rotation vector of the transformation. Has no return value.
- `getRotation`: Public member function that returns the `Point` class instance representing the rotation vector of the transformation. Takes no arguments.
- `setDisplacement`: If the Transformation instance is configured as a `Displacement`, this public member function sets the Point class instance that represents the displacement itself. Has no return value.
- `getDisplacement`: Public member function that returns the `Point` class instance representing the displacement point of the transformation. Takes no arguments.

The class has defined two operator functions that let the class be written and read to and from an input and output stream, respectively[11]. Those operators expect a normal instance, not a pointer. The next code lines represent the calls presented.

```
// Normal instance, default constructor
OpenCIF::Transformation trans;

// Manually setting the transformation values
trans.setType ( Transformation::Displacement );
trans.setDisplacement ( Point ( 10 , 10 ) );

trans.setType ( Transformation::Rotation );
trans.setRotation ( Point ( 10 , 10 ) );

trans.setType ( Transformation::VerticalMirroring );

trans.setType ( Transformation::HorizontalMirroring );

// Manually getting the transformation values
Point point;
point = trans.getRotation ();
point = trans.getDisplacement ();

// Reading a fraction from an input file
ifstream file;
/*... Opens file ...*/
file >> point;

// Writing a fraction to an output file
ofstream file;
/*... Opens file ...*/
file << point;
```

PolygonCommand

This class represents the polygon command. The polygon commands in the *CIF* format are defined by a list of points that represents the figure itself. The next list shows all the relevant member functions, constructors and destructors that the user might need.

[11]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

- `PolygonCommand`: Constructor of the class. It is overloaded, so you can call it with without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance without points. The other constructor takes as argument a C++ string. Such string must represent a complete *CIF* command with a format as defined in section 3.2. Such constructor will initialize the instance with the points defined in the string command. Both constructors set the command type as `Polygon`.
- `type`: Public member function that is polymorphic. Takes no arguments and returns a value of type `CommandType`. In the case of this class, the value returned is `Polygon`. Takes no arguments.
- `setPoints`: Public member function that helps the user to manually set a C++ vector of `Point` class instances. Such vector represents the points of the polygon itself. The call to this member function replace the previously stored points. Has no return value.
- `getPoints`: Public member function that helps the user to retrieve the points used by the polygon. The return value is a C++ vector containing `Point` class instances. Takes no arguments.

The class has defined four operator functions that let the class be written and read to and from an input and output stream, respectively[12]. Two operators works to read and write `PolygonCommand` class pointers. The other two works to read and write `PolygonCommand` class instances. The next code lines represent the calls presented.

```
// Normal instance, default constructor (no points set)
OpenCIF::PolygonCommand normal_instance;

// Pointer instance, default constructor (no points set)
OpenCIF::PolygonCommand* pointer_instance = new OpenCIF::PolygonCommand ();

// Normal instance, initializing using string
OpenCIF::PolygonCommand normal_points ( ‘‘P 0 0 100 0 100 100 ;’’ );

// Pointer instance, initializing using string
OpenCIF::PolygonCommand* pointer_points;
pointer_points = new OpenCIF::PolygonCommand ( ‘‘P 0 0 100 0 100 100 ;’’ );

// Setting points manually
vector< Point > points;
points.push_back ( Point ( 0 , 0 ) );
points.push_back ( Point ( 100 , 0 ) );
points.push_back ( Point ( 100 , 100 ) );

normal_instance.setPoints ( points );
pointer_instance->setPoints ( points );

// Getting points
vector< Point > retrieved_points;
retrieved_points = normal_points.getPoints ();
retrieved_points = pointer_points->getPoints ();

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_points;
file >> pointer_points;

// Writing instances to a file
ofstream file;
/*... Opens file ...*/
```

[12]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

```
file << normal_points;
file << pointer_points;
```

WireCommand

This class represents a wire command. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `WireCommand`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance without points and a width of 0. The other constructor takes as argument a C++ string class instance. The string must represent a valid and clean *CIF* wire command.
- `type`: Public member function that is polymorphic. Takes no arguments and returns a value of type `CommandType`. In the case of this class, the value returned is `Wire`. Takes no arguments.
- `setWidth`: Public member function that helps to set the width of the wire. Takes as argument an `unsigned long int` integer value. Has no return value.
- `getWidth`: Public member function that returns the width of the wire. Returns an `unsigned long int` integer value. Takes no arguments.
- `setPoints`: Public member function that helps the user to manually set a C++ vector of `Point` class instances. Such vector represent the points of the wire. The call to this member function replace the previously stored wire points. Has no return value.
- `getPoints`: Public member function that helps the user to retrieve the points used by the wire. The return value is a C++ vector containing `Point` class instances. Takes no arguments.

The class has, also, defined four operator functions that let the class be written and read to and from an input and output stream, respectively[13]. Two operators works to read and write `WireCommand` class pointers. The other two works to read and write `WireCommand` class instances. The next code lines represent the calls presented.

```
// Normal instance, default constructor (no points set)
OpenCIF::WireCommand normal_instance;

// Pointer instance, default constructor (no points set)
OpenCIF::WireCommand* pointer_instance = new OpenCIF::WireCommand ();

// Normal instance, initializing using string
OpenCIF::WireCommand normal_points ( "W 100 0 0 100 0 100 100 ;" );

// Pointer instance, initializing using string
OpenCIF::WireCommand* pointer_points;
pointer_points = new OpenCIF::WireCommand ( "W 100 0 0 100 0 100 100 ;" );

// Setting values manually
vector< Point > points;
points.push_back ( Point ( 0 , 0 ) );
points.push_back ( Point ( 100 , 0 ) );
points.push_back ( Point ( 100 , 100 ) );

normal_instance.setPoints ( points );
normal_instance.setWidth ( 100 );

pointer_instance->setPoints ( points );
pointer_instance->setWidth ( 100 );
```

[13]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

```
// Getting values
vector< Point > retrieved_points;
unsigned long int width;

retrieved_points = normal_points.getPoints ();
width = normal_points.getWidth ();

retrieved_points = pointer_points->getPoints ();
width = pointer_points->getWidth ();

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_points;
file >> pointer_points;

// Writing instances to a file
ofstream file;
/*... Opens file ...*/
file << normal_points;
file << pointer_points;
```

BoxCommand

This class represents a box command. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `BoxCommand`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance with a size of 1,1, a position in 0,0 and a neutral rotation of 1,0. The other constructor takes as argument a C++ string class instance. The string must represent a valid and clean *CIF* Box command.
- `type`: Public member function that is polymorphic. Takes no arguments and returns a value of type `CommandType`. In the case of this class, the value returned is `Box`. Takes no arguments.
- `setRotation`: Public member function that helps to set the rotation point of the box. The argument needed must be a `Point` class instance. Has no return value.
- `getRotation`: Public member function that returns the rotation point applied to the box. The call to this member function returns a `Point` class instance. Takes no arguments.
- `setSize`: Public member function that helps to set the size of the box. The argument needed must be a `Size` class instance. Has no return value.
- `getSize`: Public member function that returns the size of the box. The call to this member function returns a `Size` class instance. Takes no arguments.
- `setPosition`: Public member function that helps to set the position (center) of the box. The argument needed must be a `Point` class instance. Has no return value.
- `getPosition`: Public member function that returns the position of the box. The call to this member function returns a `Point` class instance. Takes no arguments.

The class has defined four operator functions that let the class be written and read to and from an input and output stream, respectively[14]. Two operators works to read and write `BoxCommand` class pointers. The other two works to read and write `BoxCommand` class instances. The next code lines represent the calls presented.

[14]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

```
// Normal instance, default constructor
OpenCIF::BoxCommand normal_instance;

// Pointer instance, default constructor
OpenCIF::BoxCommand* pointer_instance = new OpenCIF::BoxCommand ();

// Normal instance, initializing using string
OpenCIF::BoxCommand normal_points ( ‘‘B 100 100 50 50 1 0 ;’’ );

// Pointer instance, initializing using string
OpenCIF::BoxCommand* pointer_points;
pointer_points = new OpenCIF::BoxCommand ( ‘‘B 100 100 50 50 1 0 ;’’ );

// Setting values manually
normal_instance.setSize ( Size ( 100 , 100 ) );
normal_instance.setPosition ( Point ( 50 , 50 ) );
normal_instance.setRotation ( Point ( 1 , 0 ) );

pointer_instance->setSize ( Size ( 100 , 100 ) );
pointer_instance->setPosition ( Point ( 50 , 50 ) );
pointer_instance->setRotation ( Point ( 1 , 0 ) );

// Getting values
Point rotation;
Size size;
Point position;

position = normal_instance.getPosition ();
size = normal_instance.getSize ();
rotation = normal_instance.getRotation ();

position = pointer_instance->getPosition ();
size = pointer_instance->getSize ();
rotation = pointer_instance->getRotation ();

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_instance;
file >> pointer_instance;

// Writing instances to a file
ofstream file;
/*... Opens file ...*/
file << normal_instance;
file << pointer_instance;
```

RoundFlashCommand

This class represents a round flash command. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `RoundFlashCommand`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance with a diameter of 1 and a position in 0,0. The other constructor takes as argument a C++ string class instance. The string must represent a valid and clean *CIF* round flash command.
- `type`: Public member function that is polymorphic. Takes no arguments and returns a value of type

`CommandType`. In the case of this class, the value returned is `RoundFlash`. Takes no arguments.

- `setDiameter`: Public member function that helps to set the diameter of the circle. The argument needed must be an `unsigned long int` integer value. Has no return value.
- `getDiameter`: Public member function that returns the diameter of the circle. The call to this member function returns an `unsigned long int` integer value. Takes no arguments.
- `setPosition`: Public member function that helps to set the position (center) of the circle. The argument needed must be a `Point` class instance. Has no return value.
- `getPosition`: Public member function that returns the position of the circle. The call to this member function returns a `Point` class instance. Takes no arguments.

The class has defined four operator functions that let the class be written and read to and from an input and output stream, respectively[15]. Two operators works to read and write `RoundFlashCommand` class pointers. The other two works to read and write `RoundFlashCommand` class instances. The next code lines represent the calls presented.

```
// Normal instance, default constructor
OpenCIF::RoundFlashCommand normal_instance;

// Pointer instance, default constructor
OpenCIF::RoundFlashCommand* pointer_instance = new OpenCIF::RoundFlashCommand ();

// Normal instance, initializing using string
OpenCIF::RoundFlashCommand normal_instance ( "R 100 50 50 ;" );

// Pointer instance, initializing using string
OpenCIF::RoundFlashCommand* pointer_instance;
pointer_instance = new OpenCIF::RoundFlashCommand ( "R 100 50 50 ;" );

// Setting values manually
normal_instance.setDiameter ( 100 );
normal_instance.setPosition ( Point ( 50 , 50 ) );

pointer_instance->setDiameter ( 100 );
pointer_instance->setPosition ( Point ( 50 , 50 ) );

// Getting values
unsigned long int diameter;
Point position;

position = normal_instance.getPosition ();
diameter = normal_instance.getDiameter ();

position = pointer_instance->getPosition ();
diameter = pointer_instance->getDiameter ();

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_instance;
file >> pointer_instance;

// Writing instances to a file
ofstream file;
```

[15]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

```
/*... Opens file ...*/
file << normal_instance;
file << pointer_instance;
```

DefinitionStartCommand

This class represents a definition start command. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `DefinitionStartCommand`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance with and ID of 0 (zero) and a neutral AB fraction of $\frac{1}{1}$. The other constructor takes as argument a C++ string class instance. The string must represent a valid and clean *CIF* definition start command.
- `type`: Public member function that is polymorphic. Takes no arguments and returns a value of type `CommandType`. In the case of this class, the value returned is `DefinitionStart`. Takes no arguments.
- `setID`: Public member function that helps to set the ID of the definition. The argument needed must be an `unsigned long int` integer value. Has no return value.
- `getID`: Public member function that returns the ID of the definition. The call to this member function returns an `unsigned long int` integer value. Takes no arguments.
- `setAB`: Public member function that helps to set the AB fraction of the definition. The argument needed must be a `Fraction` class instance. Has no return value.
- `getAB`: Public member function that returns the AB fraction of the definition. The call to this member function returns a `Fraction` class instance. Takes no arguments.

The class has defined four operator functions that let the class be written and read to and from an input and output stream, respectively[16]. Two operators works to read and write `DefinitionStartCommand` class pointers. The other two works to read and write `DefinitionStartCommand` class instances. The next code lines represent the calls presented.

```
// Normal instance, default constructor
OpenCIF::DefinitionStartCommand normal_instance;

// Pointer instance, default constructor
OpenCIF::DefinitionStartCommand* pointer_instance = new OpenCIF::DefinitionStartCommand ();

// Normal instance, initializing using string
// The command defines an ID equal to 5 and an AB value of 1/2
OpenCIF::DefinitionStartCommand normal_instance ( "D S 5 1 2 ;" );

// Pointer instance, initializing using string
OpenCIF::DefinitionStartCommand* pointer_instance;
pointer_instance = new OpenCIF::DefinitionStartCommand ( "D S 5 1 2 ;" );

// Setting values manually
normal_instance.setID ( 5 );
normal_instance.setAB ( Fraction ( 1 , 2 ) );

pointer_instance->setID ( 5 );
pointer_instance->setAB ( Fraction ( 1 , 2 ) );

// Getting values
unsigned long int id;
```

[16]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

```
Fraction ab;

id = normal_instance.getID ();
ab = normal_instance.getAB ();

id = pointer_instance->getID ();
ab = pointer_instance->getAB ();

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_instance;
file >> pointer_instance;

// Writing instances to a file
ofstream file;
/*... Opens file ...*/
file << normal_instance;
file << pointer_instance;
```

DefinitionDeleteCommand

This class represents a definition delete command. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `DefinitionDeleteCommand`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance with and ID of 0 (zero). The other constructor takes as argument a C++ string class instance. The string must represent a valid and clean CIF definition start command.
- `type`: Public member function that is polymorphic. Takes no arguments and returns a value of type `CommandType`. In the case of this class, the value returned is `DefinitionDelete`. Takes no arguments.
- `setID`: Public member function that helps to set the ID of the definition. The argument needed must be an `unsigned long int` integer value. Has no return value.
- `getID`: Public member function that returns the ID of the definition. The call to this member function returns an `unsigned long int` integer value. Takes no arguments.

The class has defined four operator functions that let the class be written and read to and from an input and output stream, respectively[17]. Two operators works to read and write `DefinitionStartCommand` class pointers. The other two works to read and write `DefinitionStartCommand` class instances. The next code lines represent the calls presented.

```
// Normal instance, default constructor
OpenCIF::DefinitionDeleteCommand normal_instance;

// Pointer instance, default constructor
OpenCIF::DefinitionDeleteCommand* pointer_instance;
pointer_instance = new OpenCIF::DefinitionDeleteCommand ();

// Normal instance, initializing using string
OpenCIF::DefinitionDeleteCommand normal_instance ( "D D 5 ;" );

// Pointer instance, initializing using string
OpenCIF::DefinitionDeleteCommand* pointer_instance;
pointer_instance = new OpenCIF::DefinitionDeleteCommand ( "D D 5 ;" );
```

[17]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

```
// Setting values manually
normal_instance.setID ( 5 );

pointer_instance->setID ( 5 );

// Getting values
unsigned long int id;

id = normal_instance.getID ();

id = pointer_instance->getID ();

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_instance;
file >> pointer_instance;

// Writing instances to a file
ofstream file;
/*... Opens file ...*/
file << normal_instance;
file << pointer_instance;
```

CallCommand

This class represents a call command. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `CallCommand`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance with and ID of 0 (zero) and none transformation. The other constructor takes as argument a C++ string class instance. The string must represent a valid and clean *CIF* definition start command.
- `type`: Public member function that is polymorphic. Takes no arguments and returns a value of type `CommandType`. In the case of this class, the value returned is `Call`. Takes no arguments.
- `setID`: Public member function that helps to set the ID of the definition to call. The argument needed must be an `unsigned long int` integer value. Has no return value.
- `getID`: Public member function that returns the ID of the definition to call. The call to this member function returns an `unsigned long int` integer value. Takes no arguments.
- `setTransformations`: Public member function that helps to set the transformations to be applied to the call. The transformations are set using a C++ vector class instance containing `Transformation` class instances. These transformations replaces the transformations that can be already been set. Has no return value.
- `getTransformations`: Public member function that returns a vector of the transformations to be applied to the call. The returned value is a C++ vector class instance containing `Transformation` class instances. If there is none transformation configured, the vector will be empty. Takes no arguments.
- `addTransformation`: Public member function that helps to add transformations, one by one. Takes as argument a `Transformation` class instance. The transformation is added to the list of transformations of the call. Has no return value.

The class has defined four operator functions that let the class be written and read to and from an input and output stream, respectively[18]. Two operators works to read and write `CallCommand` class pointers. The other two works to read and write `CallCommand` class instances. The next code lines represent the calls presented.

[18]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

```
// Normal instance, default constructor
OpenCIF::CallCommand normal_instance;

// Pointer instance, default constructor
OpenCIF::CallCommand* pointer_instance;
pointer_instance = new OpenCIF::CallCommand ();

// Normal instance, initializing using string
OpenCIF::CallCommand normal_instance ( ‘‘C 5 R 10 10 T 100 100 M X M Y ;’’ );

// Pointer instance, initializing using string
OpenCIF::CallCommand* pointer_instance;
pointer_instance = new OpenCIF::CallCommand ( ‘‘C 5 R 10 10 T 100 100 M X M Y ;’’ );

// Setting values manually
Transformation trans;
trans.setType ( Transformation::Displacement );
trans.setDisplacement ( Point ( 100 , 100 ) );
normal_instance.setID ( 5 );
normal_instance.addTransformation ( trans );

pointer_instance->setID ( 5 );
pointer_instance->addTransformation ( trans );

// Getting values
vector< Transformation > transformations;

transformations = normal_instance.getTransformation ();

transformations = pointer_instance->getTransformation ();

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_instance;
file >> pointer_instance;

// Writing instances to a file
ofstream file;
/*... Opens file ...*/
file << normal_instance;
file << pointer_instance;
```

DefinitionEndCommand

This class represents a definition end command. The class, by itself, doesn't have any member function. It, as the previous commands, has an overloaded constructor that can be used to convert a string command into instance, but is possible to create the instance without it, since the instance itself doesn't contain any information beyond it's type (calling the `getType` member function returns a `CommandType` value, in this case, the value `DefinitionEnd`).

The class has defined four operator functions that let the class be written and read to and from an input and output stream, respectively[19]. Two operators works to read and write `DefinitionEndCommand` class pointers. The other two works to read and write `DefinitionEndCommand` class instances. The next code lines represent the calls presented.

```
// Normal instance, default constructor
```

[19]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

```
OpenCIF::DefinitionEndCommand normal_instance;

// Pointer instance, default constructor
OpenCIF::DefinitionEndCommand* pointer_instance;
pointer_instance = new OpenCIF::DefinitionEndCommand ();

// Normal instance, initializing using string
OpenCIF::DefinitionEndCommand normal_instance ( "D F ;" );

// Pointer instance, initializing using string
OpenCIF::DefinitionEndCommand* pointer_instance;
pointer_instance = new OpenCIF::DefinitionEndCommand ( "D F ;" );

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_instance;
file >> pointer_instance;

// Writing instances to a file
ofstream file;
/*... Opens file ...*/
file << normal_instance;
file << pointer_instance;
```

EndCommand

This class represents an end command. This command must be found only once in the whole design. The class, by itself, doesn't have any member function beyond it's type (calling the `getType` member function returns a `CommandType` value, in this case, the value `End`).

The class has defined four operator functions that let the class be written and read to and from an input and output stream, respectively[20]. Two operators works to read and write `EndCommand` class pointers. The other two works to read and write `EndCommand` class instances. The next code lines represent the calls presented.

```
// Normal instance
OpenCIF::EndCommand normal_instance;

// Pointer instance
OpenCIF::EndCommand* pointer_instance;
pointer_instance = new OpenCIF::EndCommand ();

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_instance;
file >> pointer_instance;

// Writing instances to a file
ofstream file;
/*... Opens file ...*/
file << normal_instance;
file << pointer_instance;
```

[20]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

UserExtensionCommand

This class represents an user extension command. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `UserExtensionCommand`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance with empty contents. The other constructor takes as argument a C++ string class instance. The string must represent a valid and clean *CIF* definition start command.
- `type`: Public member function that is polymorphic. Takes no arguments and returns a value of type `CommandType`. In the case of this class, the value returned is `UserExtention`. Takes no arguments.
- `setContents`: Public member function that helps to set the contents of the extension. The argument needed must be a string value. Has no return value.
- `getContents`: Public member function that returns the contents of the extension. The call to this member function returns a C++ string value. Takes no arguments.

The class has defined four operator functions that let the class be written and read to and from an input and output stream, respectively[21]. Two operators works to read and write `UserExtensionCommand` class pointers. The other two works to read and write `UserExtensionCommand` class instances. The next code lines represent the calls presented.

```
// Normal instance, default constructor
OpenCIF::UserExtensionCommand normal_instance;

// Pointer instance, default constructor
OpenCIF::UserExtensionCommand* pointer_instance;
pointer_instance = new OpenCIF::UserExtensionCommand ();

// Normal instance, initializing using string
OpenCIF::UserExtensionCommand normal_instance ( ‘‘9 cell_name ;’’ );

// Pointer instance, initializing using string
OpenCIF::UserExtensionCommand* pointer_instance;
pointer_instance = new OpenCIF::UserExtensionCommand ( ‘‘9 cell_name ;’’ );

// Setting values manually
normal_instance.setContents ( ‘‘9 cell_name’’ );

// Getting values
string contents;
contents = normal_instance.getContents ();

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_instance;
file >> pointer_instance;

// Writing instances to a file
ofstream file;
/*... Opens file ...*/
file << normal_instance;
file << pointer_instance;
```

[21]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

CommentCommand

This class represents a comment command. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `CommentCommand`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance with empty contents. The other constructor takes as argument a C++ string class instance. The string must represent a valid and clean CIF comment command.
- `type`: Public member function that is polymorphic. Takes no arguments and returns a value of type `CommandType`. In the case of this class, the value returned is `Comment`. Takes no arguments.
- `setContents`: Public member function that helps to set the contents of the comment. The argument needed must be a C++ string value. Has no return value.
- `getContents`: Public member function that returns the contents of the comment. The call to this member function returns a C++ string value. Takes no arguments.

The class has defined four operator functions that let the class be written and read to and from an input and output stream, respectively[22]. Two operators works to read and write `CommentCommand` class pointers. The other two works to read and write `CommentCommand` class instances. The next code lines represent the calls presented.

```
// Normal instance, default constructor
OpenCIF::CommentCommand normal_instance;

// Pointer instance, default constructor
OpenCIF::CommentCommand* pointer_instance;
pointer_instance = new OpenCIF::CommentCommand ();

// Normal instance, initializing using string
OpenCIF::CommentCommand normal_instance ( ‘‘(This is a comment) ;’’ );

// Pointer instance, initializing using string
OpenCIF::CommentCommand* pointer_instance;
pointer_instance = new OpenCIF::CommentCommand ( ‘‘(This is a comment) ;’’ );

// Setting values manually
normal_instance.setContents ( ‘‘(This is a comment)’’ );

// Getting values
string contents;
contents = normal_instance.getContents ();

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_instance;
file >> pointer_instance;

// Writing instances to a file
ofstream file;
/*... Opens file ...*/
file << normal_instance;
file << pointer_instance;
```

[22]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

LayerCommand

This class represents a layer command. The next list shows all the relevant member functions, constructors and destructors that the user might need.

- `LayerCommand`: Constructor of the class. It is overloaded, so you can call it with or without arguments. Calling without arguments will use the default constructor. The default constructor initialize the instance without name. The other constructor takes as argument a C++ string class instance. The string must represent a valid and clean CIF comment command.
- `type`: Public member function that is polymorphic. Takes no arguments and returns a value of type `CommandType`. In the case of this class, the value returned is `Layer`. Takes no arguments.
- `setName`: Public member function that helps to set the name of the layer. The argument needed must be a C++ string value. Has no return value.
- `getName`: Public member function that returns the name of the layer. The call to this member function returns a C++ string value. Takes no arguments.

The class has defined four operator functions that let the class be written and read to and from an input and output stream, respectively[23]. Two operators works to read and write `LayerCommand` class pointers. The other two works to read and write `LayerCommand` class instances. The next code lines represent the calls presented.

```
// Normal instance, default constructor
OpenCIF::LayerCommand normal_instance;

// Pointer instance, default constructor
OpenCIF::LayerCommand* pointer_instance;
pointer_instance = new OpenCIF::LayerCommand ();

// Normal instance, initializing using string
OpenCIF::LayerCommand normal_instance ( ‘‘CCP ;’’ );

// Pointer instance, initializing using string
OpenCIF::LayerCommand* pointer_instance;
pointer_instance = new OpenCIF::LayerCommand ( ‘‘CCP ;’’ );

// Setting values manually
normal_instance.setName ( ‘‘CCP’’ );

// Getting values
string contents;
contents = normal_instance.getName ();

// Loading instances from a file
ifstream file;
/*... Opens file ...*/
file >> normal_instance;
file >> pointer_instance;

// Writing instances to a file
ofstream file;
/*... Opens file ...*/
file << normal_instance;
file << pointer_instance;
```

[23]The user must be aware that the reading process depends heavily on the format of the input stream. The input stream must have a clean format as defined in the section 3.2.

3 About LibOpenCIF

In this section will be shown to the user some extra information that can be useful in some special cases. It is recommended that the user reads the previous section before checking this one.

3.1 Using LibOpenCIF in different operating systems and languages

LibOpenCIF is going to be ported to various systems. The first system is GNU/Linux and the next system must be Microsoft Windows. Finally, the last system will be Apple Mac OS. This is due to the available access to different systems that the main developer has.

It is recommended to stay checking the download page of the library in it's official SourceForge download section (SECTION HERE), since all the versions will be available there. Also, being open source, the project can be forked and anyone can perform tests to port the library to other systems as needed. In this case, is recommended to fork from the official Git project (PATH HERE).

LibOpenCIF, as indicated, is a C++ library. There is work in progress to port the library to other languages, like Python and Java. Such ports will be separated projects due to name changes (for example, in Python, the project name is expected to be pyOpenCIF), but there is expected to be references between project pages in SourceForge and Git.

3.2 Cleaning process of the string commands

The cleaning process that the library performs to the commands buffered is not complex, in fact. The process works over the fact that the commands are valid (the cleaning process comes after the validation step), so, various assumptions are being considered when cleaning.

3.2.1 Numeric commands

The numeric commands are those commands that depends only on values to form themselves. Such commands are:

- Polygon
- Box
- Wire
- Round Flash

Those commands are cleaned with the same process. The cleaning method is this:

1. Backup the identification char (P, B, W or R) on the cleaned command and add a single white space. This will create a previous and incomplete final command like this: “P ”, “W ”, “R ” or “B ”. The original command to clean is still not modified.
2. Replace, in the command to clean, all the characters that are not digits or minus chars (“-”) with whitespaces. This will erase all the separation chars in the command and replace them with white spaces. Also, the identification char will be replaced, but it is already saved in the final command. Also, the final semicolon is not saved, but is added at the end.
3. Using a C++ input string stream, all the components of the command (all the values) are extracted one by one and concatenated to the final command with a single white space at the end of such values.
4. After extracting all the values, a semicolon is concatenated to the final command.

Such process is performed really quickly since is not complex.

3.2.2 Layer commands

The layer commands are cleaned in an separated process since they need a special treatment. The layer commands works in a similar way to the numeric commands, but with some differences. The main difference is the replacement process. In the numeric commands, all the characters of the commands are replaced with white spaces, except digits and minus chars ("-"). In the case for the layers, all the characters are replaced, except uppercase characters, underscores ("_") and digits.

There is important to note that the process first save the identification char of the command (the "L" char) in the final command, and then, before replacing chars, deletes explicitly the identification char from the command to clean. This is due to the fact that such char is not going to be deleted but is not useful anymore.

As the numeric commands, after replacing chars with white spaces, the command to clean is put in an input stream and the components (the layer name) is extracted and concatenated to the final command.

3.2.3 Call commands

The call commands need their own cleaning process, as the layers do need a special process. This is due to the special and unique process needed to clean the command and to separate the components of the command that are put together (like the mirroring chars, that can be put together).

The first step of the cleaning process is to replace any character that is not a digit, upper case or minus char ("-") with white spaces.

After that, comes a special process where the components of the commands are buffered char by char, to separate the components and, after detecting the end of one component, it is concatenated to the final command.

3.2.4 Definition control commands

Such commands controls the start, end and deletion of definitions within a design. All of them are cleaned with the same process. Such process is different from the previous ones. First, the process replaces all the characters that are not digits or upper case with white spaces. After that, comes a buffering process, similar to the one find on the call command cleaning process.