XFramer 1.45 Whitepaper

MS LeRM Laboratory

www.mslerm.narod.ru

Contents

XFramer

XFramer is an extension for the programming languages C++, C# and Java. It is available as freeware. With XFramer these languages gain the ability to process frames.

Together with the compiler of the appropriate language, XFramer can be used as a frame processor (see Figure 1). For that matter, XFramer works like a preprocessor, which converts frame code in valid C++, C# or Java code, which in turn becomes processed by the appropriate compiler. The code blocks, the generated code consist of may contain textual components as well as binary ones. Textual code blocks may contain code in any target language (e.g. Basic, HTML, XML, Perl, Prolog, and so on) or even text in natural language, which can be used to generate documentation. If needed, the same programming language can be used for frame code and target language code, in the case of C++, C# or Java. It is, however, unnecessary.

The preprocessor extends the programming language by a few elements. New are the keyword `frame`, the two XFramer-macros and the embedding markers (called *flags*[1]). The macros `BEGIN_FRAME_TEXT()` and `END_FRAME_TEXT()` define the beginning and end of the target language code. Between these, the flags for the slots will be placed. These syntactical elements are identical for all the supported programming languages. Depending on the programming language used for the frame code, however, the XFramer supporting libraries *MFramer.X* and *MString.X* vary. The first library contains the so-called intrinsic methods. Intrinsic methods are methods that belong to the fixed commands of the frame processor. The other library *MString.X* provides string functionality.

The actual versions of XFramer are available on Windows and Linux from

http://www.geocities.com/mslerm/xframer

Frame Anatomy

A frame in XFramer consists of *attributes*, *methods* and a *frame text* (see Figure 2).

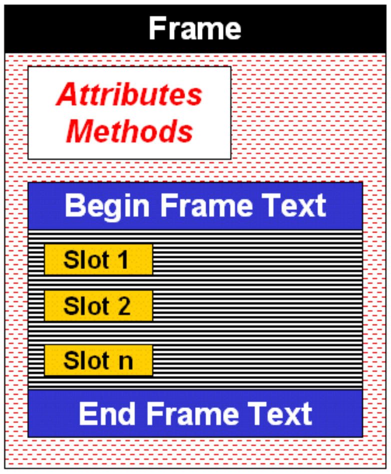

Figure 2: Frame Anatomy

The frame text contains the data, which becomes assembled and reused during the generation process (e.g. code building blocks). Any number of slots

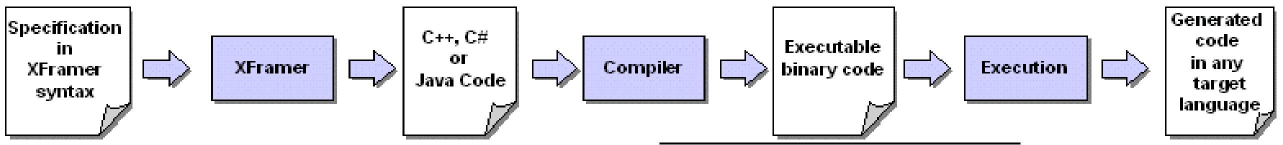
Specification in XFramer syntax
XFramer
C++, C# or Java Code
Compiler
Executable binary code
Execution
Generated code in any target language

Figure 1: Code generation with XFramer

1 The the term *flag* was chosen instead of the common term *tag* to avoid mistakes. Tags usually have a different type of syntax.

can be created inside the frame text. These slots contain variables or expressions that are either elementary or of the type `MString` or `frame`.

The first program with XFramer

The classical *Hello World* example should not be missing here (see Listing 1).

```
frame FrHelloWorld {
public:
    FrHelloWorld()
    {
BEGIN_FRAME_TEXT()
Hello World!
END_FRAME_TEXT()
    }
};
void main()
{
    FrHelloWorld frHelloworld;
    frHelloworld.exportToFile("output.txt");
}
```

Listing 1: Hello World

It is certainly impossible to demonstrate the wide range of possibilities with this example. However, the basics can be explained easily. Declarations and definition are only slightly different from those of a class. Instead of `class`, the keyword `frame` is used followed by the frame name (here: `FrHelloWorld`). It is advisable to begin all frame names with the prefix `Fr`, to visually distinguish them from the class names. In addition to the attributes and methods that are common to classes, frames contain the frame text. The macros `BEGIN_FRAME_TEXT()` and `END_FRAME_TEXT()`define the borders of the area, containing the frame text. In this example, the frame instance `frHelloWorld` is created inside the function `main` (Listing 1) with the frame text „Hello World!“. Afterwards, it is exported to the file *output.txt* by the application of the intrinsic method `exportToFile`.

To obtain the final generated source code in the target language (in this case „Hello World“ in natural language) from the specification (Listing 1), the following steps are necessary: XFramer has to be started with two parameters. The first parameter is the name of the source file (containing the specification) and the second one is the name of the destination file:

```
xframer frHelloWorld.cpp frHelloWoldFramed.cpp
```

XFramer automatically detects the language of the source file and generates a destination file which has to be compiled by the appropriate compiler. The support libraries *MString.X* and *MFramer.X* have to be in the project directory in order to perform a successful compilation. The execution of the resulting binary file *frHelloWorldFramed.exe2* produces the file *output.txt*, which contains the generated source code “Hello World”.

Building a generator with XFramer – using the example of a pizza-generator

As already mentioned at the beginning, frame technology is well suited for generative programming. Feature models [CE00; CBUE02] possess all necessary information to build a frame hierarchy.

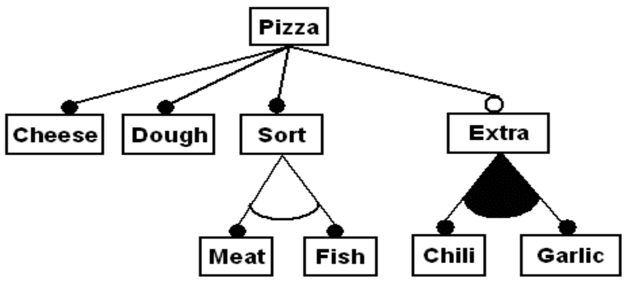

Figure 3: feature diagram of a pizza-system

Figure 3 presents a feature diagram, which models a pizza concept. *Cheese* and *Dough* are mandatory features because every pizza needs them. *Sort* is also a mandatory feature. Therefore only one feature from the group of the optional features *Fish* and *Meat* must be chosen. *Extra* is optional and has the subfeatures *Chili* and *Garlic* which are linked in an or-group.

```
frame FrTemplate {
public:
    FrTemplate(MString name)
    {
BEGIN_FRAME_TEXT()
struct <!name!> {
    <!name!>()
    {
        cout << "<!name!>" << endl;
    }
} obj<!name!>;
END_FRAME_TEXT()
    }
};
```

Listing 2: Template for the basic component of the pizza-system

Listing 2 `FrTemplate` defines a template for all elementary building blocks of the pizza system. The name is the variation point. The name defines the data type, constructor, display and object name. The realization of the feature *Sort* is shown in Listing 3.

2 If Linux is used as operating system, the file is named *frHelloWorldFramed.*

```
frame FrSort {
public:
    FrSort(MString mstr = "")
    {
BEGIN_FRAME_TEXT()
<!
(mstr != "Meat")
    ? FrTemplate("Fish")
    : FrTemplate("Meat")
!>
END_FRAME_TEXT()
    }
};
```

Listing 3: Realization of the feature „Sort“

The frame `FrSort` gets a string parameter and fills `FrTemplate`. If an empty string or invalid identifier is passed, `FrSort` fills the frame `FrTemplate` with *Fish*. The logic inside the slots can be conveniently realized with the ternary operator `?`. It is, however, not mandatory to hide the logic inside the slots, as Listing 4 shows.

```
frame FrExtra {
public:
    FrExtra(MString mstr1 = "", MString mstr2 = "")
    {
        MString mstrTemp1 = "";
        if (!mstr1.isEmpty()) {
            mstrTemp1 = (mstr1 != "Garlic")
                            ? FrTemplate("Chili")
                            : FrTemplate("Garlic");
        }

        MString mstrTemp2 = "";
        if (!mstr2.isEmpty() && mstr2 != mstr1) {
            mstrTemp2 = (mstr2 != "Garlic")
                            ? FrTemplate("Chili")
                            : FrTemplate("Garlic");
        }

BEGIN_FRAME_TEXT()
<!mstrTemp1!>
<!mstrTemp2!>
END_FRAME_TEXT()
    }
};
```

Listing 4: Realization of the feature „Extra“

`FrExtra` works in a similar way as `FrSort`. The difference is that here the components *Chili* and *Garlic* may also appear together or not at all. If the same component is used twice, a special routine is called.

```
frame FrPizza {
public:
    FrPizza(FrSort frSort, FrExtra frExtra = FrExtra())
    {
BEGIN_FRAME_TEXT()
#include <iostream.h>
<!FrTemplate("Dough")!>
<!FrTemplate("Cheese")!>
<!frSort!>
<!frExtra!>
void main(){}
END_FRAME_TEXT()
    }
};
```

Listing 5: Pizza concept-frame

`FrPizza` (see Listing 5) is the root of the whole pizza frame hierarchy. The constructor of `FrPizza` accepts the frame instances of `FrSort` and optionally `FrExtra`. The instantiation of `FrTemplate` embeds the components *Cheese* and *Dough*, which are mandatory parts of the pizza system. Subsequently, the *Sort* and *Extra* instances are embedded.

```
void main()
{
    FrPizza(FrSort()).exportToFile("pizza1.cpp");

    FrPizza(FrSort("Fish"),
      FrExtra("Chili")).exportToFile("pizza2.cpp");

    FrPizza(FrSort("Meat"), FrExtra("Garlic",
      "Chili")).exportToFile("pizza3.cpp");
}
```

Listing 6: Pizza orders

Finally the Listing 6 shows how various pizzas can be assembled. The instances of the frame `FrPizza` contain a C++-program that the intrinsic function `exportToFile` exports to a file, which in turn is compiled by a C++ compiler.

Embedding of files

Another feature of XFramer is the possibility to embed files. In the case, that existing code has to be encapsulated in frames manually. This can be time consuming and would render the code more complex than necessary. In the case of binary files, there are even additional problem to solve.

```
1. BEGIN_FRAME_TEXT()
2. <#example.bin#>
3. <!MString().importFromFile("example.bin")!>
4. END_FRAME_TEXT()
```

Listing 7: Pizza concept-frame

Therefore, XFramer has the file embedding flags <# and #> (see listing 7 line 2). Inside these flags a file with path may be declared, which becomes embedded at this place. Please notice the similarity to a C++ *include*-directive.

Files can however, also be embedded inside of slots (<! !>). This is possible by the application of the `importFromFile` method of `MString` class (see listing 7, line 3). In the latter case, the file content becomes loaded not until runtime of the generator as opposed to embedding flags, which weaves the contents, like resources into the generator.

Parameterization of frame blocks

In some cases XFramer-flags can become in conflict with characters of the target language. An example is the generation of HTML-pages. There is a conflict between the HTML-comment characters <!—and the slot flag <!.

```
BEGIN_FRAME_TEXT(FLAG_<! = "<BEGIN_OF_EMBEDDING>",
                 FLAG_!> = "<END_OF_EMBEDDING >")
<!-- this is a comment --!>
<BEGIN_OF_EMBEDDING>
Slot
<END_OF_EMBEDDING>
END_FRAME_TEXT()
```

Listing 8: Renaming of the slot flags

Fortunately, this problem could be solved easily by renaming slot flags (see listing 8). XFramer-directives, which begin with the prefix `FLAG_`, allow the assignment of a new string. These flag substitutions are optional and can be announced between the brackets of the `BEGIN_FRAME_TEXT`-macro. For that matter, the order of the assignments is irrelevant. The validity of the flags substitutions is local and limited to the following frame text block.

This local implementation in XFramer was consciously preferred to the usual solution with configuration file, because of the following reasons.

- The replacement can always be easily spotted in the code, therefore improving maintains.
- Other projects can reuse the code without alteration, due to the substitutions are contained in the code itself.
- The various frame text parameterization can be applied anywhere in any number.

References

[CE00] K. Czarnecki and U. W. Eisenecker, Generative Programming - Methods, Tools, and Applications, Addison-Wesley, 2000, see:: http://www.generative-programming.org

[CBOE02] K. Czarnecki, T. Bednasch, U. W. Eisenecker, P. Unger, Generative Programming for Embedded Software: An Industrial Experience Report, In online proceedings of the GCPE 2002