[Codegs in Cyan]

CODEGS: VISUAL METAOBJECTS OF LANGUAGE CYAN

A PREPRINT

André Souza
andre.cassulino@gmail.com

José de Oliveira Guimarães
Department of Computer Science at Sorocaba
Federal University of São Carlos at Sorocaba - SP, Brazil
josedeoliveiraguimaraes@gmail.com

August 10, 2023

ABSTRACT

Nowadays, programs are described mainly through texts, which is adequate for most situations. However, many code snippets and even whole programs are better described visually as those involving colors, text formatting, diagrams, and Graphical User Interfaces (GUI). A mixture of textual and visual programming is offered by the Cyan language through Codegs, which are compile-time metaobjects. Codegs can interact with the Integrated Development Environment during editing time and, with the data collected using a GUI, generate code and do checks during compilation time. Besides that, Codegs allow interactions between the source code and external tools such as the command prompt and batch files.

***K*eywords** Object-oriented languages, DSL, metaprogramming, compiler, IDE, visual languages, metaobject protocols

1 Introduction

This paper draws ideas from two fields: Graphical User Interfaces (GUI) and metaprogramming. GUIs are largely used for getting user input through menus, buttons, windows, drop-down lists, etc. Integrated Development Environments (IDE) employ GUI to blend text editors, compilers, debuggers, and other development tools. IDEs may use GUI to generate code based on input taken visually. For example, a GUI can generate code for other GUI [1].

Metaprogramming is the handling of programs by programs. This paper will consider only Compile-Time Metaprogramming (CTMP), which is the handling of a program at compile-time. CTMP may be achieved by means of a Metaobject Protocol (MOP), which is a protocol that describes the interactions between the program, the compiler, and the metaprogram. The later is a program that describes the transformations or checks that the program should suffer. It acts like a compiler plugin that changes the way the program is compiled.

Notwithstanding the advances that occurred in the last decades in the design of programming languages, programming continues to be largely a one-dimensional activity, made in a text editor. This works gives a contribution in order to change this state of affairs. It combines Graphical User Interfaces with Metaprogramming in an original way.

This article describes *Codegs*, which are visual metaobjects of the prototype-based object-oriented language Cyan. Prototypes are the counterpart of classes of more traditional languages as Java and C♯. A metaobject is an object of the metaprogram that exists only at compile-time. It is created because of an annotation like

```
@color(red)
```

in *the base program*. Methods of metaobjects are called by the compiler in order to do additional checks, beyond those made by the compiler, and add code to the current prototype.

The main contribution of this paper is the description of *Codegs*, a variation of metaobjects. Their annotations are triggered at editing time, in the IDE, when the user hovers the mouse over them. The metaobject associated with the annotation opens up a window from which data can be got visually. This data is used later at compile-time to produce code, inserted in the current prototype, or do additional checks in the program. In this way, Graphical User Interfaces can be used for a great variety of common programming tasks without the need of building an IDE plugin.

Listing 1: Example with annotations

```
1  package main
2
3  object Person
4      var String name
5      var String country
6      @init(name, country)
7      @insertCode{*
8          for str in [ "name", "country" ] {
9              var s = "func get_" ++ str ++ " -> String ";
10             insert: s, s ++ " = " ++ str ++ "; ";
11         }
12     *}
13     func run {
14         var p = Person new: "Isaac Newton", "England";
15         Out println: p get_name;
16         Out println: "This line number is " ++ @lineNumber;
17     }
18 end
```

This article is organized as follows: Section 2 describes metaobjects in Cyan. Codegs themselves are explained in Section 3. Works related to Codegs are discussed in Section 4. The last Section concludes, as always.

2 Metaobjects in Cyan

Metaprogramming in Cyan is made through a compile-time Metaobject Protocol (MOP) which allows the interception of field (instance variable) access, method call, method overriding, and inheritance of a prototype. But not only this: through the MOP, fields and methods may be added to prototypes and statements and expressions to methods. Additional checks can be made. All of this at compile-time.

The Cyan Metaobject Protocol describes the relationships among the base program, its metaprogram, and the compiler. The base program is compiled usually unless the compiler finds an "annotation" which, in this article, will always have the format

```
@annotName(params){* Text *}
```

`annotName` is the annotation name or just "annotation" and `params` are parameters which may be a comma-separated list of literals (integers, strings, identifiers, literal arrays, etc.). Finally, `Text` is any text, normally code of a Domain Specific Language (DSL). Both the parameters, with `(` and `)`, and the DSL code are optional. Annotations link the program and the metaprogram. For each annotation, the compiler creates a single metaobject during parsing. This metaobject will be responsible to add code to the prototype and do checks. The annotation belongs to the program and its associated metaobject to the metaprogram. Before explaining how the compilation is changed by a metaobject, let us learn more on annotations.

Annotations may be attached to a declaration such as a prototype, method, field, or local variable. Some annotations may be used as expressions and others where a statement would be expected. This is best explained using an example, shown in Listing 1.

Annotation `init` in line 6 will cause the addition of a *constructor* for prototype `Person` to initialize fields `name` and `country`. A prototype is a literal object that plays the role of a class in class-based languages such as Java, C♯, and C++. Because of annotation `init`, an object of `Person` can be created as shown in line 14 using method `new:`. Cyan employs a syntax for method declaration and message passing similar to the Smalltalk language. Annotation `insertCode` in line 7 has an attached DSL code which is interpreted Cyan, also called *Tyan*. Language *Tyan* allows only statements and is dynamically typed. The DSL statement `insert:`[1] of line 10 produces Cyan code that is inserted in prototype `Person`. Two methods are produced, `get_name` and `get_country`. Method `++` converts the message receiver and its parameter to strings, concatenates them, and return the result. Method `insert:` takes two parameters: the signature of the method (or field) to be inserted and the full method (or the field with, possibly, the value assigned to it). The signature does not have the method body. In line 15, method `get_name` of object `p` is called and the result is printed. In line 16, annotation `lineNumber` is used as an expression. The code it produces, `16`, will be the value printed.

[1] This is a message to `self` that refers to an object that has an `insert:` method.

There is a one-to-one correspondence between annotations and metaobjects. For that reason, we will say "metaobject `init`" instead of "metaobject associated to annotation `init`" if there is just one `init` annotation. Each metaobject is an object of a Java *metaobject class* or a Cyan prototype that has a method `getName` that returns, as a string, the annotation name. The compiled form of a metaobject class or prototype should be put in a package. When the package is imported, the annotations become available just as the package prototypes. In the example of Listing 1, all annotations are associated to metaobject classes (in Java) of package `cyan.lang`, automatically imported by every Cyan source code.

Metaobjects can be made in Java or in Cyan. Java is preferred because the compiler is made in Java, making trivial the communications between the compiler and metaobjects. Metaobject classes in bytecodes[2] of the Java Virtual Machine are dynamically loaded when a package is imported.

The Cyan compiler has six main compilation phases, precisely described elsewhere [2]. The compiler builds the Abstract Syntax Tree (AST) in phase 1. In phase 2 the compiler associates types to fields, method parameters and return value types, and inherited prototypes. When the AST is built, it has only the names of the types, as strings. Phase 2 associates a reference to a prototype for every element that has a type and is outside a method. Then only types external to method bodies are typed in phase 2, the semantic analysis of method statements is not done.

In phase 3, metaobjects may ask the compiler to insert fields and methods to prototypes. The metaobject associated with an annotation can only insert code in the prototype in which the annotation is. Code is inserted as strings in memory only, the original source files are not changed. This demands the recompilation of the source file in memory since it was modified. Semantic analysis of method statements is made in phase 4. During this phase, statements and expression can be inserted, causing the recompilation of the source file. Checks are preferably made in phase 5 when the source file cannot be changed anymore. Phase 6 is code generation to Java, the same language of the compiler and the metaobject classes.

The important phases for this article are 3 and 4. Metaobjects `insertCode` and `init` of Listing 1 insert methods in phase 3 and metaobject `lineNumber` of line 16 inserts code in phase 4.

Each metaobject class should implement one or more Java interfaces according to its goals. If it intends to insert fields and methods, it should implement interface `IAction_afti`, as the classes of metaobjects `init` and `insertCode` do. A method of this interface returns code to be inserted in the current prototype. The code is added in phase 3.

Similarly, if the intention is to insert code inside a method, as a statement or expression, the metaobject should implement interface `IAction_dsa`, as the class of `lineNumber` does. This code insertion is made in phase 4.

This Section presented a simplified vision of the Cyan Metaobject Protocol. A complete description is in the language site [2].

3 Codegs, Visual Metaobjects

Codegs are metaobjects that have all the power of regular metaobjects and that can get user input at editing time. They are supported by an Eclipse IDE [3] plugin downloadable from the site of the language [2]. Before going into the details, we will show how they work through an example, that of Codeg `color`. As before, it will be used "Codeg `color`" for "Codeg whose annotation is `color`" or "Codeg whose method `getName` returns `"color"`".

At editing time in the Eclipse IDE with the plugin for Codegs, suppose the mouse hovers over word "`color`" in the line

```
Out println: @color(red)
```

A window opens up as shown in Figure 1. The user can then choose the color visually, after which she pushes button "Ok". The window disappears and the editing can continue. The annotation parameter `red` is just an identifier, its relation to the chosen color is not checked. This parameter can be changed either using the text editor or in the Codeg window. Every Codeg annotation should have at least one parameter that identifies it univocally within the prototype.

When prototype `Program` of Figure 1 is compiled, the Codeg `color` retrieves the color chosen at editing time and produces it as code. The result, in this particular example, is the same as if "`@color(red)`" were replaced by `16711680` (FF0000 in Hex). Let us explain what happens in details.

The Codeg plugin to the Eclipse IDE calls the Cyan compiler every few milliseconds after the user stopped typing. But only to parse the source file being edited. This is enough to import metaobjects from the imported packages (including `cyan.lang`, imported by every source file). When the mouse is over the text, the plugin will discover whether the

[2]Prototypes in Cyan are compiled into Java classes.

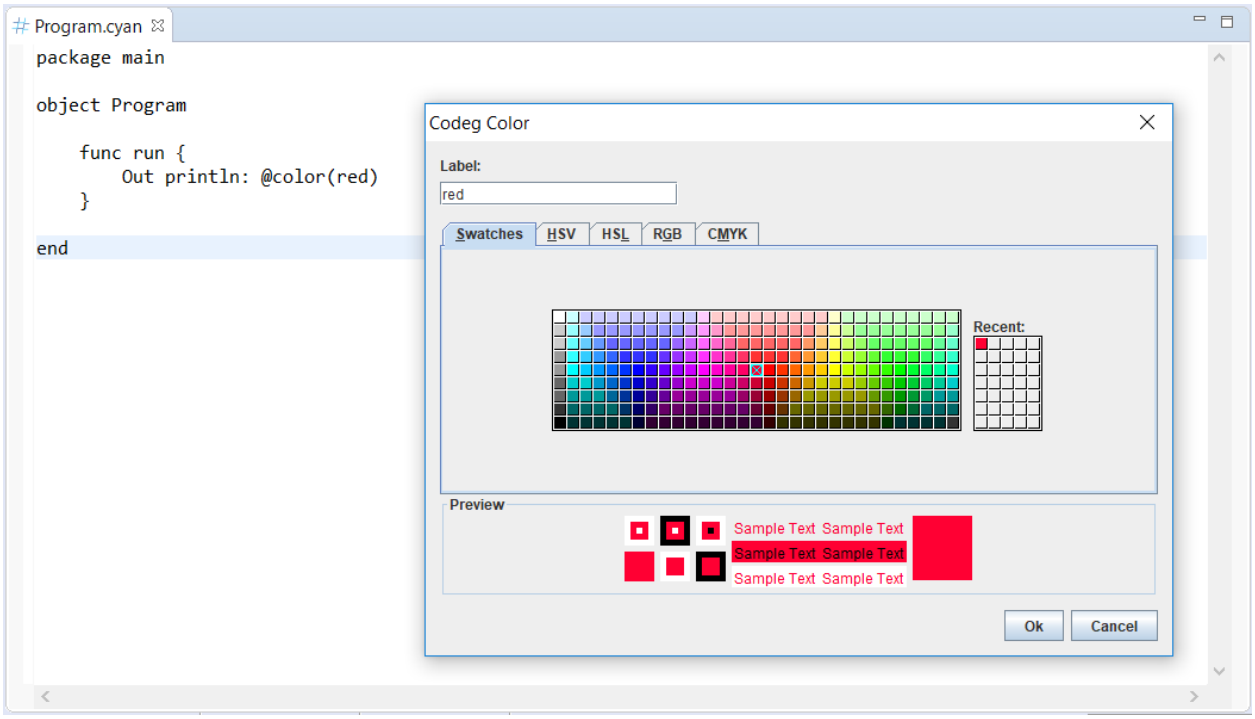

Figure 1: Codeg `color`

mouse pointer is over a Codeg annotation or not. The plugin knows where are the Codegs in the text because the compiler, after parsing the text being edited, passes a list with all Codegs to the plugin. Each Codeg keeps its starting and ending offset in the text.

If the mouse is over a Codeg annotation, the plugin calls a compiler method with the Codeg as a parameter and, in its turn, this method calls `getUserInput` of the Codeg. This method is declared in interface `ICodeg` that should be implemented by every Codeg class.

```
byte [] getUserInput( ICompiler_ded compiler_ded,
                      byte [] previousCodegFileText);
```

Method `getUserInput` should open a window for getting user input visually. It is expected that it has buttons "Ok" and "Cancel". The data gathered through the GUI should be converted into a byte array and returned. The first parameter to `getUserInput` gives to the Codeg a restricted view of the compiler. The second parameter is `null` if the method is being called for the first time or the byte array returned by the previous call to the method.

Using parameter `compiler_ded`, the Codeg has access to information on the prototype in which the annotation is such as the list of instance variables and local variables visible in the point of the annotation. At editing time, the source being edited was only parsed, there is no semantic information.

The compiler method that calls `getUserInput` saves the returned byte array in a file in the directory of the package of the current prototype. The file name is composed by the prototype name, the Codeg name, and the identifier. So it is unique. The class of Codeg `color` implements interface `IAction_dsa` in order to generate code in phase 6 of the compilation. To generate code it needs the color number chosen at editing time. This color number, as a byte array, is returned from method `getCodegInfo` inherited from a class that is a superclass of all Codeg classes. The same byte array returned by `getUserInput` is returned by `getCodegInfo`. Between the editing time and the compilation, the byte array is kept in the file just described. In the general case, the data gathered at editing time will be converted to a DSL code that, as a byte array, will be returned by `getUserInput`.

Codegs can communicate with each other at editing time. To enable that, a Codeg class should implement a Java interface. This interface has a method that should return the data the Codeg wants to share. And another to receive data from all other Codegs whose annotations are in the same prototype. Then a Codeg may access data collected from the GUI of other Codegs in the same prototype. For example, there could several Codegs for defining GUI elements

such as buttons, windows, panels, menus, etc. And a super Codeg that is able to manage all GUI elements of the other Codegs. The other Codegs would share their information with the super Codeg.

A Codeg class can enable communication in compilation phases 3, 4, and 5 by implementing another interface. The workings are the same as with communication at editing time. A Codeg annotation may have more than one parameter, the single restriction is that the first one should be a valid identifier. The other may be literals of arrays, maps, tuples, etc. A DSL code may be attached to the annotation too. This code can even be changed by the GUI of the Codeg.

4 Related Work

Graphical user interfaces are used in IDE tools for many goals, such as code generation for GUI themselves [1]. These tools are generally plugins to the IDE. Unlike an IDE plugin, the GUI of a Codeg demands a Cyan source file be used, its use is more limited in this aspect. But Codegs are also metaobjects and they have all the advantages of this feature. A metaobject can add fields and methods to the prototype where its annotation is, generate code that is added after its annotation[3], intercept field access, method calls, inheritance of a prototype, do checks using any AST object it has access to, etc. A metaobject has access to the AST object of the annotation it is attached to, the current method, prototype, package, etc. A `visit` method supplied by the compiler implements the *Visitor* Design Pattern. Through `visit`, the metaobject can do checks in the program. For example, metaobject `immutable` checks if its attached prototype is immutable; that is, if its fields cannot be changed after the object creation.

Visual Languages [4] [5] allow the specification of programs through non-textual means. Codegs have a more limited goal, they are a complement to the Cyan language. But in the extreme case, a Codeg could implement a full Visual Language that generates Cyan code.

Codea [6] is an IOs App for developing games. It has an editor for the Lua language [7] in which a color may be chosen much like our Codeg `color`. The same can be made with images and sounds, as the Codegs `image` and `sound` of our Codeg library. To our knowledge, the only Codeg-related features are the three just cited, they cannot be user-made.

The IntelliJ IDEA IDE of Jetbrains supports a feature called *language injection* [8] that makes it easy to add code of a language inside another. The example that follows is in Java.

```
@Language("HTML")
String s = "<body> Hi! </body>";
```

Annotation `Language` takes, as a parameter, a language name. The IDE assumes that the following string keeps a text that is code of this language (HTML in this example). Then the literal string of the next line can be edited with IDE help such as syntax highlighting and code completion. It may also be edited in a separated *fragment editor*. Language injection can be user-defined.

Language injections have different goals in relation to Codegs. They cannot generate code or do checks in the program, their help is in editing time only. Codegs do not offer, by themselves, any editor support like language injections.

Metaobjects are supported by many languages such as OpenC++ [9], OJ [10], and CLOS [11]. However, we are unaware of any language in which they are integrated with a GUI or even an IDE.

5 Conclusion

Three Codegs were cited in this paper: `color`, `image`, and `sound`. Many others were made. Codeg `batch` opens a window with a text editor for a DOS batch program. The initial directory for execution can also be chosen. The result of the execution is shown in a box. Codeg `cmd` opens a command prompt in a user-chosen directory. It is for convenience only, no code is generated. Codeg `foundFile` opens a window that allows the user to choose a file in the file system. A string with the file name is returned, the annotation should be used as an expression (as `color`). Codeg `re` takes a regular expression as input and tests it with several expressions picked out by the user. The programmer can then check if the regular expression matches her/his expectations. There is a Codeg for video playing too. A graph can be visually defined by Codeg `graph`. The call order of methods of a prototype are specified using a Finite State Machine (FSM) in Codeg `fsmMethods`. This Codeg employs several metaobject features because it produces fields and methods for the prototype and code that is added to methods. Finally, Codeg `cyan` opens up a text window in which Cyan statements can be typed and interpreted. The output goes to another windows. There is a *live* mode in which the statements are interpreted while the user is typing them.

[3] This is the case of metaobject `color`.

Codegs are a combination of compile-time metaobjects and Graphical User Interfaces, all supported by an IDE plugin. They are especially useful for getting input that is more easily given visually. The input got by a GUI may be used for code generation and checks at compile-time. Since Codegs are metaobjects, they can use all the power of the Cyan Metaobject Protocol.

Codegs can also be viewed as a tool for building (limited) Eclipse plugins. What is needed is just the implementation of a metaobject class with a `getUserInput` method. The Codeg plugin takes care of all other interactions with Eclipse.

References

[1] Windowbuilder, july 2018.

[2] José de Oliveira Guimarães. The cyan language. 2020.

[3] Eclipse. The eclipse ide, September 2018.

[4] Kang Zhang. *Visual Languages and Applications*. Springer-Verlag, Berlin, Heidelberg, 2007.

[5] Margaret M. Burnett. Visual languages. In *Encyclopedia of Eletrical and Eletronics Engineering*, pages 275–283. John Wiley & Sons Inc., New York, 1999.

[6] Codea. Codea, july 2018.

[7] Roberto Ierusalimschy. *Programming in Lua, Third Edition*. Lua.Org, 3rd edition, 2013.

[8] IDEA. Using language injections, 2018.

[9] Shigeru Chiba. A metaobject protocol for c++. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 285–299, New York, NY, USA, 1995. ACM.

[10] Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. Openjava: A class-based macro system for java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering, Papers from OORaSE 1999*, pages 117–133, London, UK, UK, 2000. Springer-Verlag.

[11] Andreas Paepcke. Object-oriented programming. In Andreas Paepcke, editor, *Object-oriented Programming*, chapter User-level Language Crafting: Introducing the CLOS Metaobject Protocol, pages 65–99. MIT Press, Cambridge, MA, USA, 1993.