Protein Programming

Philippe Larvet, Research engineer, *Alcatel-Lucent Villarceaux, December 2007*

***Abstract*—**Within the context of application building with object-oriented programming (which is today the standard for application development) this paper presents a new and simpler way to facilitate / automate the writing of classes and applications. This simple way is the use of a code generator based on the concept of a Reduced Instruction Set Programming (RISP) driven by only 4 types of instructions : "access", "compute", "test" and "generate" (i.e A, C, T, G, the four amino-acid pieces that build proteins).

I. What is the problem to solve?

Within the context of modern application building with object-oriented programming [1], (which is today the standard for application development [5]) the technical problem to solve is to facilitate / to automate the writing of classes and applications by the easiest way as possible.

The best existing solution today is to write the application by using software development environments (Java SDK, Eclipse, Visual Studio, etc) and making big efforts to write the code of classes [3], [4].

II. What is the main idea of the solution we propose?

A long practice of application programming leads us to observe that only four kind of instructions are useful to write the code of a given program, no matter how complex it is: we need *accessors* to get data and values stored within the classes of the application; we have to *compute* those data, in order to produce new values and results – which are ordinary the main objective of the application; we need *tests*, in order to evaluate the results we get or the next step of the algorithm, and to determine the new action to trigger; and finally we need to *generate* events from a given class to others, in order to execute the procedures and methods embedded in these classes. See also Steve Mellor [7], p.125.

So, the necessity of these four kinds of actions is closer to the structure of biological proteins. Proteins, and also DNA – desoxy-ribonucleic acid [2] – are assemblies of four - and only four - pieces, the amino-acids Adenine, Cytosine, Thymine and Guanine (A, C, T, G) [6].

According to this natural disposition, the basic idea of the *Protein Programming solution* is to propose a code generator based on a Reduced Instruction Set Programming (RISP) driven by only four types of instructions:

- A for "access", in order to access the data stored in the classes of the application;
- C for "compute", in order to calculate values/results to be stored inside classes or to be exchanged between classes;
- T for "test", in order to evaluate conditions and determine what new actions have to be done;
- G for "generate", in order to generate events to other classes.

So, an application can be seen as – an application *is* – a set of RISP instructions A, C, T, G that can be taken as inputs by a code generator able to build a complete and compilable application from only these four types of statements.

III. Description of one embodiment of the solution

Within the context of RISP "protein programming", an application can be seen as a "polypeptide", made of different classes (the "peptides") that store data and communicate by events.

An event is a call of a public method of a class, eventually carrying parameters. In the biological world, proteins communicate by molecular exchanges, through specific receptors and actuators.

The RISP protein code generator uses only 4 instructions, expressed as predicates: A() for accessors, C() for computations, T() for tests and G() for generating events.

1. A = access. When a peptide (a class) wants to access the molecule (data) "data_name" stored in the other peptide "class_name", the Accessor function is used: `A(class_name, data_name);`
2. C = calculate. When a new molecule (a calculation) is needed, the Compute function is used: : `C(result, operand1, operator, operand2);`
3. T = test. The evaluation of a condition is done with the T() function : `T(condition, action_if_true, action_if_false);`
4. G = generate. When a peptide (a class) wants to activate the function "method_name" of another peptide "class_name" with the molecule "parameter", the function G() is used : `G(class_name, method_name, parameter);`

This notation is the entry point for the protein code generator, that produces the final code of the application, in Java for example.

This RISP ACTG notation, simple and generic, is a help for building applications, as well as for learning object-oriented

Philippe Larvet was research engineer at Alcatel-Lucent Bell Labs, Villarceaux (France). Author of numerous publications and patents, he works now as independent consultant in Bergerac (France). e-mail: phlarvet@gmail.com. LinkedIn https://www.linkedin.com/in/philippe-larvet-9ba391/

programming and teaching the basic principles of object-oriented development [6]. The notation allows describing an application (a polypeptide) through its sequencing, like a DNA or a protein sequencing (ACTG-AACG-ACCG-ACTG-etc.).

IV. Example of a simple Protein Programming Application

In order to illustrate our solution, we have chosen the simple problem of using an ATM (automated teller machine), as it is described in Wikipedia : see the following link http://en.wikipedia.org/wiki/Automated_teller_machine)

"An automated teller machine (ATM) is a computerized telecommunications device that provides the customers of a financial institution (a bank) with access to financial transactions in a public space without the need for a human clerk or bank teller. On most modern ATMs, the customer is identified by inserting a plastic card with a magnetic stripe or a plastic smartcard with a chip that contains a unique card number and some security information, such as an expiration date. Security is provided by the customer entering a personal identification number (PIN). Using an ATM, customers can access their bank accounts in order to make cash withdrawals (or credit card cash advances) and check their account balances".

Following this example, we can describe a simple protein-application able to allow a customer to make a withdrawal, according to the following use case:

- the customer introduces his card in the ATM
- the ATM asks the customer to enter his secret code (PIN code)
- the customer enters his code
- the ATM (card reader) checks the code
- if the code is wrong, after 3 tries, the card is returned to the user
- if the code is correct, the ATM asks the customer to enter the amount of his withdrawal
- the customer enters the amount of money he wishes to get
- the ATM checks onto the card if the withdrawal amount is authorized
- the ATM, via a bank interface, asks the customer's account if the bank balance allows the withdrawal
- if these conditions are OK, the banknotes can be distributed

The analysis of this use case gives us the active classes composing the application (the "polypeptide"): card, card reader, bank interface, ATM, distribution slot.

The exchanges of events between these elements allow to implement the use case:

1. every "peptide" (every class) has its own chain of "amino-acids" (the instructions A, C, T, or G) describing its functions (the methods of the class);
2. the assembly of all these peptides compose a "protein" = a "polypeptide" = an application.

Within the context of the ATM example, the formal descriptions of some "peptides" in terms of ACTG notation are depicted in the following table.

TABLE I

Formal description, in ACTG notation, of some "Peptides" for the ATM Application example

```
card {
   data:
      secret_code=1234;
}

card_reader {
   data:
      code_input;

   methods:
      introduce_card() {
        G(dialogue, "Your card is being read");
        G(card_reader, check_code);
      }

      check_code() {
        G(dialogue, "Please enter your PIN code");
        C(code_input, input());
        T(code_input==A(card, secret_code),
G(code_OK), G(code_KO) );
      }

      code_OK {
        G(ATM, ask_amount);
      }

      code_KO() {
        G(dialogue, "Wrong code");
        G(card_reader, check_code);
      }

      return_card() {
        G(dialogue, "Please take your card to get
your money");
        G(slot, distribute, A(ATM,
withdrawal_amount) );
      }
}

slot {
   data:
      nb_of_banknotes;

   methods:
      distribute(parameter) {
        C(nb_of_banknotes=parameter/A(banknote,
value) );
        G(dialogue,"Here are your money (nb of
banknotes = "+nb_of_banknotes+")" );
      }
}
```

From the ACTG notation, the protein code generator produces the Java class code for the "peptide" card-reader that is shown in Table II.

TABLE II
JAVA CLASS GENERATED BY PROTEIN GENERATOR FROM THE DESCRIPTION IN TABLE I

```
class card_reader {
   //data:
      int code_input;

   // instances (generated)
      dialogue thedialogue;
      card thecard;
      ATM theATM;
      slot theslot;

   //constructor (generated)
      public card_reader() {
        thedialogue = new dialogue();
        thecard = new card();
        theATM = new ATM();
        theslot = new slot();
      }

   //methods:
      public void introduce_card() {
        thedialogue.display("Your card is being
read");
        this.check_code();
      }

      public void check_code() {
        thedialogue.display("Please enter your PIN
code");
        code_input = input();
        if(code_input==thecard.secret_code)
          this.code_OK();
        else
          this.code_KO();
      }

      public void code_OK() {
        theATM.ask_amount();
      }

      public void code_KO() {
        thedialogue.display("Wrong code");
        this.check_code();
      }

      public void return_card() {
        thedialogue.display("Please take your card
to get your money");
        theslot.distribute(
          theATM.withdrawal_amount);
      }
}
```

V. DIFFERENCES WITH OTHER BIOLOGICAL ALGORITHMIC APPROACHES

In order to distinguish the proposed solution from the state-of-the-art, we indicate here some differences with other biological approaches like genetic algorithms, neural networks and populationist ant-programming.

V.1 Genetic algorithms [8]

Genetic algorithms (or evolutionary algorithms) belong to the family of meta-heuristic algorithms, whose the aim is to get a close solution, in an acceptable time, to an optimization problem, when no exact method is known to solve the problem in a reasonable time. By using the concepts of gene and mutation, genetic algorithms use the notion of natural selection and evolution, developed during the XIX century by Charles Darwin, and apply these concepts to a population of potential solutions to the given problem. So, the processing gets closer, by successive "jumps", to an acceptable solution.

Our solution is different because we don't use the concept of gene, nor mutation, nor natural selection, nor evolution, and the aim is not an optimization but a means to produce more easily an executable program within the scope of object-oriented programming.

V.2 Artificial Neural networks [9]

An Artificial Neural Network is a computation model whose the design is very schematically inspired from the functionning of true neurons (human or not). The neural networks are generally optimized by statistic-type learning methods, so they are located firstly in the family of statistical applications, which they enrich with a set of paradigms allowing to generate wide functional spaces, flexible and partially structured, and secondly in the family of Artifical Intelligence (AI) methods, which they enrich by allowing to take decisions that lean more on the perception than on the formal logical reasoning.

Our solution is different because we don't use the concept of neuron, nor statistic, and the Protein Programming is not an AI method: the aim is not a possibility to fire a decision, but a means to ease the production of executable programs within the scope of object-oriented programming.

V.3 Ant programming [10]

The works on ant-programming develop this concept with the goal of gaining a deeper understanding on ant colony optimization, a heuristic method for combinatorial optimization problems inspired by the foraging behavior of ants.

Indeed, ant programming allows a deeper insight into the general principles underlying the use of an iterated Monte Carlo approach for the multi-stage solution of a combinatorial optimization problem. Such an insight is intended to provide the designer of algorithms with new categories, an expressive terminology, and tools for dealing effectively with the peculiarities of the problem at hand. Ant-programming searches for the optimal policy of a multi-stage decision problem to which the original combinatorial problem is reduced.

Our solution is different because we don't use the concept of ant, nor colony, nor optimization. Our aim is not an heuristic for combinatorial optimization problems, but just a means to ease the production of executable programs within the scope of object-oriented programming.

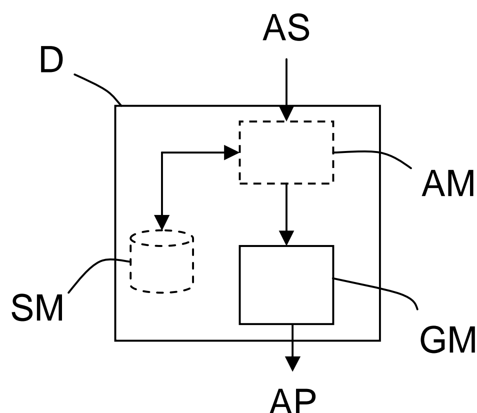
VI. CONCLUSION

We have presented in this paper an approach and a simple notation that allow developing engineers to program an application without writing complex lines of code, by using a very reduced set of instructions. Indeed, only four types of instructions are necessary to write any kind of program or complex application.

From this work is born a patent: “*Device and method for building compilable and executable applications from specifications expressed by classes*”, US 20090178023 A1.

The main concept developed in this patent is the following:

A device (D) is intended for building compilable and executable applications (AP) from high-level representations of classes, each class storing data and/or implementing at least one public function and/or being able to activate at least one chosen public function of at least one other class. This device (D) comprises a generation means (GM) arranged i) for producing “new” class representations in a chosen programming language from formal representations of specification classes expressing a specification (AS) describing an application (AP) to be built, each class formal representation being written in a high-level symbolic language comprising a class declaration, a data declaration, a function declaration, and a restricted group of instruction types chosen among four basic types comprising respectively instructions for accessing a chosen stored data of a chosen class, instructions for computing a chosen data from a chosen operator and possibly from some given input parameter(s), instructions for testing if a chosen class data satisfies to a chosen condition, and instructions for generating an activation of a chosen public function of any class possibly with at least one chosen data parameter, and ii) for assembling these new class representations to build a compilable and executable application (AP) corresponding to the specification (AS).



REFERENCES

[1] John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, ISBN 0-521-78098-5, p.278. Lists: Dynamic dispatch, abstraction, subtype polymorphism, and inheritance.

[2] J. D. Watson et F. H. C. Crick, *Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid*, Nature, vol. 171, no 4356, 25 avril 1953, p. 737-738 (PMID 13054692, DOI 10.1038/171737a0, Bibcode 1953Natur.171..737W.

[3] John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, ISBN 0-521-78098-5, p.278.

[4] Pierce, Benjamin, *Types and Programming Languages,* MIT Press, 2002, ISBN 0-262-16209-1, section 18.1 "What is Object-Oriented Programming?"

[5] Grady Booch, *Object-Oriented Analysis and Design With Applications,* Addison-Wesley, ISBN 0-8053-5340-2, l5th Printing, December 1998.

[6] Nomenclature Committee of the International Union of Biochemistry (NC-IUB), *Nomenclature for Incompletely Specified Bases in Nucleic Acid Sequences* [archive], sur IUBMB [archive], 1984.

[7] Sally Shlaer, Stephen J. Mellor, *Object Lifecycles, Modeling the World in States,* Yourdon Press Computing Series, Prentice Hall, 1992, ISBN 0-13-629940-7 p.125, *Forming and Assigning Processes.*

[8] Eiben, A. E. et al, *Genetic algorithms with multi-parent recombination*. PPSN III: Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: 78–87. ISBN 3-540-58484-6., 1994

[9] S. Hochreiter., *Untersuchungen zu dynamischen neuronalen Netzen*, Diploma thesis. Institut f. Informatik, Technische Univ. Munich. Advisor: J. Schmidhuber, 1991.

[10] Yuehui Chen & Ajith Abraham, *Tree-Structure Based Hybrid Computational Intelligence,* Intelligent Systems Reference Library, Springer-Verlag Berlin Heidelberg, 2006, pp. 121 et suivantes.

[11] Paul T. Ward, Stephen J. Mellor, *Structured Development for Real-Time Systems*, Yourdon Press, Prentice Hall, 1986, ISBN 0-13-854787-4, Vol.I, Introduction & Tools.