

The cognitive programming paradigm - the next programming structure

Benjamin Odei Bempong

Information and Communication Technology/Mathematics Department Presbyterian University College, Ghana. Abetifi-Kwahu, Eastern Region

Emailaddress:

benbodei@presbyuniversity.edu.gh / benbodei@yahoo.com (B. Odei Bempong)

To cite this article:

Benjamin Odei Bempong. The Cognitive Programming Paradigm the Next Programming Structure, *American Journal of Software Engineering and Applications*. Vol. 2, No. 2, 2013, pp. 54-67. doi: 10.11648/j.ajsea.20130202.15

Abstract: The development of computer programming started with the development of the switching logic, because, the computer hardware is made up of millions of digital switches. The activation and deactivation of these switches are through codified instructions – (program) which trigger the switches to function. The computer programming languages have gone through a revolution, from the machine code language, through assembly mnemonics to the high level programming languages like FORTRAN, ALGOL, COBOL, LISP, BASIC, ADA and C/C⁺⁺. It is a fact that, these programming languages are not the exact codes that microprocessors do understand and work with, because through compiler and interpreter programs, these high level programming languages that are easily understood by people are converted to machine code languages for the microprocessor to understand and do the work human knowledge has instructed it to do. The various programming languages stem from the difficulties man has in using one programming language to solve different problems on the computer. Hence, for mathematical and trigonometric problems, FORTRAN is the best, for business problems, COBOL is the right language, whilst for computer games and designs, BASIC language is the solution. The trend of using individual programming languages to solve specific problems by single processor computers have changed drastically, from single core processors to present day dual and multi-core processors. The main target of engineers and scientists is to reach a stage that the computer can think like the human brain. The human brain contains many cognitive (thinking) modules that work in parallel to produce a unique result. With the presence of multi-core processors, why should computers continue to draw summaries from stored databases, and allow us to sit hours to analyse these results to find solutions to problems? The subject of ‘Cognitive Programming Paradigm’, analyses the various programming structures and came out that these programming structures are performing similar tasks of processing stored databases and producing summarized information. These summarized information are not final, business managers and Executives have to sit hours to deliberate on what strategic decisions to take. Again, present day computers cannot solve problems holistically, as normally appear to human beings. Hence, there’s the need for these programming structures be grouped together to solve human problems holistically, like the human brains processing complex problems holistically. With the presence of multi-core processors, its possible to structure programming such that these programming structures could be run in parallel to solve a specific problem completely, i.e. be able to analyse which programming structure will be suitable for a particular problem solving or be able to store first solution and compare with new solutions of a problem to arrive at a strategic decision than its being done at present. This approach could lift the burden on Managers and Executives in deliberating further on results of a processed business problem.

Keywords: Programming; Structure; Cognitive; Paradigm

1. Introduction

The Computer technology which started with the switching logic and the discovery of binary decimal has gone through an evolution especially in designing artificial language that can talk to electro-digital systems. In process, a

number of programming structures have emerged, that dictated the type of electro-digital devices available at the time to be communicated to. There has been a big struggle therefore between Computer Hardware designers, and their counter parts - the software designers. One may ask what are these struggles about? At one time Hardware designers

design a system that puts a pressure on Software designers. The Software designers think around it and develop something that challenges the hardware capabilities, thus reducing what earlier the Hardware designers had done. These struggles have made the Computer technology one of the fastest disciplines the world had seen. But is it the end...?. It is the author's believe that with the current breakthrough in technology in terms of dual-core and multi-core processors, high speed processors and the 64-bit word length processors, the challenge has been shifted once again to the Software developers.

2. Types of Programming Structures

In this paper, literature reviews and analysis will be drawn on various programming structures, their importance, advantages and disadvantages, and brought closer to the workings of the human brain, for which these programming paradigms have been prototyped for, but which, it seems, a great omission has been left out, not avertdedly, but due to failure of mankind to satisfy both sides of software and hardware developments. In recent times there have been uncountable programming languages whose review would be difficult, however, these programming languages could be grouped under specific programming structures or paradigms that have emerged- structured, procedural, functional, logic, declarative, imperative, abductive logic, metalogic, constraint logic, concurrent logic, inductive logic and Higher-order logic programming. These would be compared with the functions of the human brain to determine if we are getting our programs structures perfect.

2.1. Structured Programming Languages

Structured programming is a programming paradigm aimed on improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures and for and while loops - in contrast to using simple tests and jumps such as the 'goto' statement which could lead to "spaghetti code" which was both difficult to follow and to maintain.

It emerged in the 1960s, particularly from work by Böhm and Jacopini, and a famous letter, "Go To Statement Considered Harmful", from Edsger Dijkstra in 1968 [1] and was bolstered theoretically by the structured program theorem, and practically by the emergence of languages such as ALGOL with suitably rich control structures. These led to the development of low level structured programming languages. [4]

2.2. Low-level Structure Programming

At a low level, structured programs are often composed of simple, hierarchical program flow structures. These are sequence, selection, and repetition:

"Sequence" refers to an ordered execution of statements.

In "selection" one of a number of statements is executed depending on the state of the program. This is usually ex-

pressed with keywords such as `if..then..else..endif`, `switch`, or `case`.

In "repetition" a statement is executed until the program reaches a certain state, or operations have been applied to every element of a collection. This is usually expressed with keywords such as `while`, `repeat`, `for` or `do..until`. Often it is recommended that each loop should only have one entry point (and in the original structural programming, also only one exit point, and a few languages enforce this).

A language is described as block-structured when it has a syntax for enclosing structures between bracketed keywords, such as an `if`-statement bracketed by `if..fi` as in ALGOL 68, or a code section bracketed by `BEGIN..END`, as in PL/I - or the curly braces `{...}` of C and many later languages.

It is possible to do structured programming in any programming language, though it is preferable to use something like a procedural programming language. Some of the languages initially used for structured programming languages include: ALGOL, Pascal, PL/I and Ada - but most new procedural programming languages since that time have included features to encourage structured programming, and sometimes deliberately left out features that would make unstructured programming easy.

Historically, the structured program theorem provides the theoretical basis of structured programming. It states that three ways of combining programs—sequencing, selection, and iteration—are sufficient to express any computable function. This observation did not originate with the structured programming movement; these structures are sufficient to describe the instruction cycle of a central processing unit, as well as the operation of a Turing machine. Therefore a processor is always executing a "structured program" in this sense, even if the instructions it reads from memory are not part of a structured program. However, authors usually credit the result to a 1966 paper by Böhm and Jacopini, possibly because Dijkstra cited this paper himself. The structured program theorem does not address how to write and analyze a usefully structured program. These issues were addressed during the late 1960s and early 1970s, with major contributions by Dijkstra, Robert W. Floyd, Tony Hoare, and David Gries. [4]

There has been serious debates on structured programming, in which the early users like P. J. Plauger, an early adopter of structured programming, described his reaction to the structured program theorem and said that they have been converted to adopt structured programming but for assembly-language programming it was found difficult to be structured, and neither these were proved by Bohm and Jacopini nor their repeated successes at writing structured code brought them round one day sooner than they were ready to convince themselves that structured programming are different from assembly-language programming.

Donald Knuth accepted the principle that programs must be written with provability in mind, but he disagreed with abolishing the GOTO statement. In his 1974 paper, "Structured Programming with Goto Statements", he gave exam-

ples where he believed that a direct jump leads to clearer and more efficient code without sacrificing provability. Knuth proposed a looser structural constraint: It should be possible to draw a program's flow chart with all forward branches on the left, all backward branches on the right, and no branches crossing each other. Structured programming theorists gained a major ally in the 1970s after IBM researcher Harlan Mills applied his interpretation of structured programming theory to the development of an indexing system for the New York Times research file. The project was a great engineering success, and managers at other companies cited it in support of adopting structured programming, although Dijkstra criticized the ways that Mills's interpretation differed from the published work.

As late as 1987 it was still possible to raise the question of structured programming in a computer science journal. Frank Rubin did so in that year with a letter, "GOTO considered harmful" considered harmful." Numerous objections followed, including a response from Dijkstra that sharply criticized both Rubin and the concessions other writers made when responding to him.

I used to program in Assembly Language using Motorola 68000 at the University, and I can confirm the above difficulties being explained by these writers, that Assembly programming is not and cannot be structured. If one makes a mistake, going back and changing addresses is a hell. In structured programming language like BASIC, one could easily insert a statement between two line statements, and go away, but in an Assembly Language it's not that simple. One has to remember each statement and the number of address bytes each command uses before the correction could be effective. Failure to include remarks in Assembly Language programming comes the difficult for the programmer to remember what the one had done.

The results of these debates came out at the end of the 20th century when nearly all computer scientists were convinced that it is useful to learn and apply the concepts of structured programming. High-level programming languages that originally lacked programming structures, such as FORTRAN, COBOL, and BASIC, now have them. At this point let us look at the various programming structures, starting from Declarative programming:

2.3. Declarative programming

In Comparing programming paradigms, Dr. Rachel Harrison and Mr. Lins Samaraweera described Declarative programming as when one writes a program code in such a way that it describes what one wants to do, and not what one wants to do it. It is left up to the compiler to figure out how. Examples are Structured Query Language(SQL) and Prolog. But even in Prolog, it's only the logical style that look declarative. Prolog is a language that clearly breaks away from the how-type languages, encouraging the programmer to describe situations and problems not the detailed means by which the problems are to be solved. These two programs work with databases and cannot be used to compute business logic. Declarative is 'what of' program-

ming and Imperative is the 'how'. Therefore Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow. It is a programming paradigm where problems are described, or conditions on a solution are described, and the computer finds a solution. Often it involves the separation of "facts" from operations on the facts. "Declarative Programming" has also been described as a model of computation that "generalizes the pure functional model". That is, all pure functional programs are declarative programs, and Declarative Programming retains almost all of the advantages of Functional Programming. This is explained in more detail in the book Concepts Techniques And Models Of Computer Programming.

There are subcategories of Declarative Programming that include Logic Programming, Constraint Programming, and Constraint Logic Programming (which, as its name suggests, is a hybrid of the two - see Constraint And Logic Programming). While the above is nice and easy way, it is vague and leads to lots of arguments about whether this or that language is 'declarative' or effectively supports 'Declarative Programming'. Due to the vagueness of the original definition, Declarative Programming is effectively a sliding scale. The following operational characteristics might be used to judge the extent to which a programming model is declarative:

In Declarative Programming, order of statements and expressions should not affect program semantics. For example, they should not affect the termination characteristics and should not affect the observable IO properties, and ideally shouldn't affect performance.

In Declarative Programming, replication of a statement should not affect program semantics. What matters is that some fact or constraint exists, not how many times it is declared or computed. This allows refactoring and abstraction of statements, and is also a basis for many optimizations.

Many languages applying this style attempt to minimize or eliminate side effects by describing what the program should accomplish, rather than describing how to go about accomplishing it. This is in contrast with imperative programming, which requires an explicitly provided algorithm.

Declarative programming often considers programs as theories of a formal logic, and computations as deductions in that logic space. Declarative programming has become of particular interest recently, as it may greatly simplify writing parallel programs.[3]

Common declarative languages include those of regular expressions, logic programming, and functional programming.

2.4. Functional Programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state.[1]

Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus.[1]

In practice, the difference between a mathematical function and the notion of a "function" used in imperative programming is that imperative functions can have side effects, changing the value of program state. Because of this they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program. Conversely, in functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result $f(x)$ both times. Eliminating side effects can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.[1]

Functional programming languages, especially purely functional ones, have largely been emphasized in academia rather than in commercial software development. Programming in a functional style can also be accomplished in languages that aren't specifically designed for functional programming. For example, the imperative Perl programming language has been the subject of a book describing how to apply functional programming concepts.[23]

2.5. Imperative Programming

This is a programming paradigm that describes computation in terms of statements that change a program state. In much the same way that imperative mood in natural languages expresses commands to take action, imperative programs define sequences of commands for the computer to perform.

The term is used in opposition to declarative programming, which expresses what the program should accomplish without prescribing how to do it in terms of sequences of actions to be taken. Functional and logical programming are examples of a more declarative approach.

2.6. Logic Programming

This, in its broadest sense, is the use of mathematical logic for computer programming. In this view of logic programming, which can be traced at least as far back as Alonzo Church[1932], logical inference can be used in programming. This view was further developed by John McCarthy's [1958] advice-taker proposal to use forward chaining under the control of logical propositions. The Planner programming language [1969, 1971] used both forward chaining (invoked by assertions) and backward chaining (invoked by goals).

However, Kowalski [1973] restricted logic programming to backwards chaining in the form:

G if G_1 and ... and G_n

that treats the implications as goal-reduction procedures: to show/solve G , show/solve G_1 and ... and G_n .

For example, it treats the implication:

The drive cab is alerted if an alarm signal button is pressed.

as a procedure that from the goal "the drive cab is alerted" generates the sub-goal "an alarm signal button is pressed."

Note that this is consistent with the BHK interpretation of constructive logic, where implication would be interpreted as a solution of problem G given solutions of G_1 ... G_n .

The defining feature of logic programming is that sets of formulas can be regarded as programs and proof search can be given a computational meaning. In some approaches the underlying logic is restricted, e.g., Horn clauses or Hereditary Harrop formulas.

As in the purely declarative case, the programmer is responsible for ensuring the truth of programs. But since automated proof search is generally infeasible, logic programming as commonly understood also relies on the programmer to ensure that inferences are generated efficiently. In many cases, to achieve efficiency, one needs to be aware of and to exploit the problem-solving behavior of the theorem-prover. In this respect, logic programming is comparable to conventional imperative programming; using programs to control the behavior of a program executor. However, unlike conventional imperative programs, which have only a procedural interpretation, logic programs also have a declarative, logical interpretation, which helps to ensure their correctness. Moreover, such programs, being declarative, are at a higher conceptual level than purely imperative programs; and their program executors, being theorem-provers, operate at a higher conceptual level than conventional compilers and interpreters.

Historically, Logic programming in the first and wider sense gave rise to a number of implementations, such as those by Fischer Black (1964), James Slagle (1965) and Cordell Green (1969), which were question-answering systems in the spirit of McCarthy's Advice taker. Foster and Elcock's Absys (1969), on the other hand, was probably the first language to be explicitly developed as an assertional programming language.

Logic programming in the narrower sense can be traced back to debates in the late 1960s and early 1970s about declarative versus procedural representations of knowledge in Artificial Intelligence. Advocates of declarative representations were notably working at Stanford, associated with John McCarthy, Bertram Raphael and Cordell Green, and in Edinburgh, with J. Alan Robinson (an academic visitor from Syracuse University), Pat Hayes, and Robert Kowalski. Advocates of procedural representations were mainly centered at MIT, under the leadership of Marvin Minsky and Seymour Papert.

Although it was based on logic, Planner, developed at

MIT, it was the first language to emerge within this proceduralist paradigm [Hewitt, 1969]. Planner featured pattern directed invocation of procedural plans from goals (i.e. forward chaining) and from assertions (i.e. backward chaining). The most influential implementation of Planner was the subset of Planner, called Micro-Planner, implemented by Gerry Sussman, Eugene Charniak and Terry Winograd. It was used to implement Winograd's natural-language understanding program SHRDLU, which was a landmark at that time. In order to cope with the very limited memory systems that were available when it was developed, Planner used backtracking control structure so that only one possible computation path had to be stored at a time. From Planner were developed the programming languages QA-4, Popler, Conniver, QLISP, and the concurrent language Ether.

Hayes and Kowalski in Edinburgh tried to reconcile the logic-based declarative approach to knowledge representation with Planner's procedural approach. Hayes (1973) developed an equational language, Golux, in which different procedures could be obtained by altering the behavior of the theorem prover. Kowalski, on the other hand, showed how SL-resolution treats implications as goal-reduction procedures. Kowalski collaborated with Colmerauer in Marseille, who developed these ideas in the design and implementation of the programming language Prolog.

2.7. Abductive Logic Programming

Abductive Logic Programming is an extension of normal Logic Programming that allows some predicates, declared as abducible predicates, to be incompletely defined. Problem solving is achieved by deriving hypotheses expressed in terms of the abducible predicates as solutions of problems to be solved. These problems can be either observations that need to be explained (as in classical abductive reasoning) or goals to be achieved (as in normal logic programming). It has been used to solve problems in Diagnosis, Planning, Natural Language and Machine Learning. It has also been used to interpret Negation as Failure as a form of abductive reasoning.

2.8. Metalogic Programming

Because mathematical logic has a long tradition of distinguishing between object language and metalanguage, logic programming also allows metalevel programming. The simplest metalogic program is the so-called "vanilla" meta-interpreter:

```
solve(true).

solve((A,B)):- solve(A),solve(B).

solve(A):- clause(A,B),solve(B).
```

where true represents an empty conjunction, and clause(A,B) means there is an object-level clause of the form $A :- B$.

Metalogic programming allows object-level and metale-

vel representations to be combined, as in natural language. It can also be used to implement any logic that is specified by means of inference rules.

2.9. Constraint Logic Programming

Constraint logic programming is an extension of normal Logic Programming that allows some predicates, declared as constraint predicates, to occur as literals in the body of clauses. These literals are not solved by goal-reduction using program clauses, but are added to a store of constraints, which is required to be consistent with some built-in semantics of the constraint predicates.

Problem solving is achieved by reducing the initial problem to a satisfiable set of constraints. Constraint logic programming has been used to solve problems in such fields as civil engineering, mechanical engineering, digital circuit verification, automated timetabling, air traffic control, and finance. It is closely related to abductive logic programming.

2.10. Concurrent Logic Programming

Keith Clark, Steve Gregory, Vijay Saraswat, Udi Shapiro, Kazunori Ueda, etc. developed a family of Prolog-like concurrent message passing systems using unification of shared variables and data structure streams for messages. Efforts were made to base these systems on mathematical logic, and they were used as the basis of the Japanese Fifth Generation Project (ICOT). However, the Prolog-like concurrent systems were based on message passing and consequently were subject to the same indeterminacy as other concurrent message-passing systems, such as Actors (see Indeterminacy in concurrent computation). Consequently, the ICOT languages were not based on logic in the sense that computational steps could not be logically deduced [Hewitt and Agha, 1988].

Concurrent constraint logic programming combines concurrent logic programming and constraint logic programming, using constraints to control concurrency. A clause can contain a guard, which is a set of constraints that may block the applicability of the clause. When the guards of several clauses are satisfied, concurrent constraint logic programming makes a committed choice to the use of only one.

2.11. Inductive Logic Programming

Inductive logic programming is concerned with generalizing positive and negative examples in the context of background knowledge. Generalizations, as well as the examples and background knowledge, are expressed in logic programming syntax. Recent work in this area, combining logic programming, learning and probability, has given rise to the new field of statistical relational learning and probabilistic inductive logic programming.

2.12. Higher-Order Logic Programming

Several researchers have extended logic programming

with higher-order programming features derived from higher-order logic, such as predicate variables. Such languages include the Prolog extensions HiLog and λ Prolog.

2.13. Linear Logic Programming

Basing logic programming within linear logic has resulted in the design of logic programming languages that are considerably more expressive than those based on classical logic. Horn clause programs can only represent state change by the change in arguments to predicates. In linear logic programming, one can use the ambient linear logic to support state change. Some early designs of logic programming languages based on linear logic include LO [Andreoli & Pareschi, 1991], Lolli [Hodas & Miller, 1994], ACL [Kobayashi & Yonezawa, 1994], and Forum [Miller, 1996]. Forum provides a goal-directed interpretation of all of linear logic.

2.14. Procedural Programming

Procedural programming can sometimes be used as a synonym for imperative programming (specifying the steps the program must take to reach the desired state), but can also refer (as in this article) to a programming paradigm, derived from structured programming, based upon the concept of the procedure call. Procedures, also known as routines, subroutines, methods, or functions (not to be confused with mathematical functions, but similar to those used in functional programming) simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including other procedures or itself. In order to understand this procedural programming a comparison will be made with other programming languages.

2.15. Object-Oriented Programming

The Wikipedia, the free encyclopedia defines Object-oriented programming (OOP) as a programming paradigm using "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance. Many modern programming languages now support OOP, at least as an option.

Simple, non-OOP programs may be one "long" list of statements (or commands). More complex programs will often group smaller sections of these statements into functions or subroutines each of which might perform a particular task. With designs of this sort, it is common for some of the program's data to be 'global', i.e. accessible from any part of the program. As programs grow in size, allowing any function to modify any piece of data means that bugs can have wide-reaching effects.

In contrast, the object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the program. Instead, the data is accessed by

calling specially written functions, commonly called methods, which are either bundled in with the data or inherited from "class objects." These act as the intermediaries for retrieving or modifying the data they control. The programming construct that combines data with a set of methods for accessing and managing those data is called an object. The practice of using subroutines to examine or modify certain kinds of data, however, was also quite commonly used in non-OOP modular programming, well before the widespread use of object-oriented programming.

An object-oriented program will usually contain different types of objects, each type corresponding to a particular kind of complex data to be managed or perhaps to a real-world object or concept such as a bank account, a hockey player, or a bulldozer. A program might well contain multiple copies of each type of object, one for each of the real-world objects the program is dealing with. For instance, there could be one bank account object for each real-world account at a particular bank. Each copy of the bank account object would be alike in the methods it offers for manipulating or reading its data, but the data inside each object would differ reflecting the different history of each account.

Objects can be thought of as wrapping their data within a set of functions designed to ensure that the data are used appropriately, and to assist in that use. The object's methods will typically include checks and safeguards that are specific to the types of data the object contains. An object can also offer simple-to-use, standardized methods for performing particular operations on its data, while concealing the specifics of how those tasks are accomplished. In this way alterations can be made to the internal structure or methods of an object without requiring that the rest of the program be modified. This approach can also be used to offer standardized methods across different types of objects. As an example, several different types of objects might offer print methods. Each type of object might implement that print method in a different way, reflecting the different kinds of data each contains, but all the different print methods might be called in the same standardized manner from elsewhere in the program. These features become especially useful when more than one programmer is contributing code to a project or when the goal is to reuse code between projects.

Object-oriented programming has roots that can be traced to the 1960s. As hardware and software became increasingly complex, manageability often became a concern. Researchers studied ways to maintain software quality and developed object-oriented programming in part to address common problems by strongly emphasizing discrete, reusable units of programming logic. The technology focuses on data rather than processes, with programs composed of self-sufficient modules ("classes"), each instance of which ("objects") contains all the information needed to manipulate its own data structure ("members"). This is in contrast to the existing modular programming that had been dominant for many years that focused on the function of a module, rather than specifically the data, but equally provided for code reuse, and self-sufficient reusable units of pro-

gramming logic, enabling collaboration through the use of linked modules (subroutines). This more conventional approach, which still persists, tends to consider data and behavior separately.

An object-oriented program may thus be viewed as a collection of interacting objects, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent "machine" with a distinct role or responsibility. The actions (or "methods") on these objects

Object-oriented programming developed as the dominant programming methodology in the early and mid 1990s when programming languages supporting the techniques became widely available. These included Visual FoxPro 3.0,[10][11][12] C++, and Delphi. Its dominance was further enhanced by the rising popularity of graphical user interfaces, which rely heavily upon object-oriented programming techniques. An example of a closely related dynamic GUI library and OOP language can be found in the Cocoa frameworks on Mac OS X, written in Objective-C, an object-oriented, dynamic messaging extension to C based on Smalltalk. OOP toolkits also enhanced the popularity of event-driven programming (although this concept is not limited to OOP). Some feel that association with GUIs (real or perceived) was what propelled OOP into the programming mainstream.

Object-oriented features have been added to many existing languages during that time, including Ada, BASIC, Fortran, Pascal, and others. Adding these features to languages that were not initially designed for them often led to problems with compatibility and maintainability of code.

More recently, a number of languages have emerged that are primarily object-oriented yet compatible with procedural methodology, such as Python and Ruby. Probably the most commercially important recent object-oriented languages are Visual Basic.NET (VB.NET) and C#, both designed for Microsoft's .NET platform, and Java, developed by Sun Microsystems. Both frameworks show the benefit of using OOP by creating an abstraction from implementation in their own way. VB.NET and C# support cross-language inheritance, allowing classes defined in one language to subclass classes defined in the other language. Developers usually compile Java to bytecode, allowing Java to run on any operating system for which a Java virtual machine is available. VB.NET and C# make use of the Strategy pattern to accomplish cross-language inheritance, whereas Java makes use of the Adapter pattern.

3. Comparing Imperative, Procedural, and Declarative programming

Procedural programming is imperative programming in which the program is built from one or more procedures (also known as subroutines or functions). The terms are

often used as synonyms, but the use of procedures has a dramatic effect on how imperative programs appear and how they are constructed. Heavily procedural programming, in which state changes are localized to procedures or restricted to explicit arguments and returns from procedures, is known as structured programming. From the 1960s onwards, structured programming and modular programming in general, have been promoted as techniques to improve the maintainability and overall quality of imperative programs. Object-oriented programming extends this approach.

Procedural programming could be considered as a step towards declarative programming. A programmer can often tell, simply by looking at the names, arguments and return types of procedures (and related comments), what a particular procedure is supposed to do - without necessarily looking at the detail of how the procedure achieves its result. At the same time, a complete program is still imperative since it 'fixes' the statements to be executed and their order of execution to a large extent.

Declarative programming is a non-imperative style of programming in which programs describe the desired results of the program, without explicitly listing command or steps that need to be carried out to achieve the results. Functional and logical programming languages are characterized by a declarative programming style.

In a pure functional language, such as Haskell, all functions are without side effects, and state changes are only represented as functions that transform the state. Although pure functional languages are non-imperative, they often provide a facility for describing the effect of a function as a series of steps. Other functional languages, such as Lisp, OCaml and Erlang, support a mixture of procedural and functional programming.

Programs written in logical programming languages, consist of logical statements, and the program executes by searching for proofs of the statements. As in functional programming languages, some logical programming languages such as Prolog, and database query languages such as SQL, while declarative in principle, also support a procedural style of programming.

Many imperative programming languages (such as Fortran, BASIC and C) were abstractions of assembly language.

The hardware implementation of almost all computers is imperative; nearly all computer hardware is designed to execute machine code, which is native to the computer, written in the imperative style. From this low-level perspective, the program state is defined by the contents of memory, and the statements are instructions in the native machine language of the computer. Higher-level imperative languages use variables and more complex statements, but still follow the same paradigm. Recipes and process checklists, while not computer programs, are also familiar concepts that are similar in style to imperative programming; each step is an instruction, and the physical world holds the state. Since the basic ideas of imperative programming are both conceptually familiar and directly embodied in the

hardware, most computer languages are in the imperative style.

Assignment statements, in imperative paradigm, perform an operation on information located in memory and store the results in memory for later use. High-level imperative languages, in addition, permit the evaluation of complex expressions, which may consist of a combination of arithmetic operations and function evaluations, and the assignment of the resulting value to memory. Looping statements (such as in while loops, do while loops and for loops) allow a sequence of statements to be executed multiple times. Loops can either execute the statements they contain a pre-defined number of times, or they can execute them repeatedly until some condition changes. Conditional branching statements allow a sequence of statements to be executed only if some condition is met. Otherwise, the statements are skipped and the execution sequence continues from the statement following them. Unconditional branching statements allow the execution sequence to be transferred to some other part of the program. These include the jump (called "goto" in many languages), switch and the subprogram, or procedure, call (which usually returns to the next statement after the call).

3.1. Prolog

The programming language Prolog was developed in 1972 by Alain Colmerauer. It emerged from a collaboration between Colmerauer in Marseille and Robert Kowalski in Edinburgh. Colmerauer was working on natural language understanding, using logic to represent semantics and using resolution for question-answering. During the summer of 1971, Colmerauer and Kowalski discovered that the clausal form of logic could be used to represent formal grammars and that resolution theorem provers could be used for parsing. They observed that some theorem provers, like hyper-resolution, behave as bottom-up parsers and others, like SL-resolution (1971), behave as top-down parsers.

It was in the following summer of 1972, that Kowalski, again working with Colmerauer, developed the procedural interpretation of implications. This dual declarative/procedural interpretation later became formalised in the Prolog notation

$$H :- B_1, \dots, B_n.$$

which can be read (and used) both declaratively and procedurally. It also became clear that such clauses could be restricted to definite clauses or Horn clauses, where H , B_1 , ..., B_n are all atomic predicate logic formulae, and that SL-resolution could be restricted (and generalised) to LUSH or SLD-resolution. Kowalski's procedural interpretation and LUSH were described in a 1973 memo, published in 1974.

Colmerauer, with Philippe Roussel, used this dual interpretation of clauses as the basis of Prolog, which was implemented in the summer and autumn of 1972. The first Prolog program, also written in 1972 and implemented in

Marseille, was a French question-answering system. The use of Prolog as a practical programming language was given great momentum by the development of a compiler by David Warren in Edinburgh in 1977. Experiments demonstrated that Edinburgh Prolog could compete with the processing speed of other symbolic programming languages such as Lisp. Edinburgh Prolog became the *de facto* standard and strongly influenced the definition of ISO standard Prolog.

3.2. Comparison with other Programming Languages

Procedural programming languages are also imperative languages, because they make explicit references to the state of the execution environment. This could be anything from variables (which may correspond to processor registers) to something like the position of the "turtle" in the Logo programming language.

With Object-Oriented Programming, the focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines, whereas in object-oriented programming it is to break down a programming task into data types (classes) that associate behavior (methods) with data (members or attributes). The most important distinction is whereas procedural programming uses procedures to operate on data structures, object-oriented programming bundles the two together so an "object", which is an instance of a class, operates on its "own" data structure. Nomenclature therefore varies between the two, although they have similar semantics:

Procedural	Object-oriented
procedure	method
record	object
module	class
procedure call	message

In comparing Procedural programming with functional programming, the principles of modularity and code reuse in practical functional languages are fundamentally the same as in procedural languages, since they both stem from structured programming. So for example:

Procedures correspond to functions. Both allow the reuse of the same code in various parts of the programs, and at various points of its execution.

By the same token, procedure calls correspond to function application.

Functions and their invocations are modularly separated from each other in the same manner, by the use of function arguments, return values and variable scopes.

The main difference between the styles is that functional programming languages remove or at least deemphasize the imperative elements of procedural programming. The feature set of functional languages is therefore designed to support writing programs as much as possible in terms of

pure functions:

Whereas procedural languages model execution of the program as a sequence of imperative commands that may implicitly alter shared state, functional programming languages model execution as the evaluation of complex expressions that only depend on each other in terms of arguments and return values. For this reason, functional programs can have a freer order of code execution, and the languages may offer little control over the order in which various parts of the program are executed. (For example, the arguments to a procedure invocation in Scheme are executed in an arbitrary order.)

Functional programming languages support (and heavily use) first-class functions, anonymous functions and closures.

Functional programming languages tend to rely on tail call optimization and higher-order functions instead of imperative looping constructs.

Many functional languages, however, are in fact impurely functional and offer imperative/procedural constructs that allow the programmer to write programs in procedural style, or in a combination of both styles. It is common for input/output code in functional languages to be written in a procedural style.

There do exist a few esoteric functional languages (like Unlambda) that eschew structured programming precepts for the sake of being difficult to program in (and therefore challenging). These languages are the exception to the common ground between procedural and functional languages.

In logic programming, a program is a set of premises, and computation is performed by attempting to prove candidate theorems. From this point of view, logic programs are declarative, focusing on what the problem is, rather than on how to solve it.

However, the backward reasoning technique, implemented by SLD resolution, used to solve problems in logic programming languages such as Prolog, treats programs as goal-reduction procedures. Thus clauses of the form:

$$H :- B_1, \dots, B_n.$$

have a dual interpretation, both as procedures to show/solve H , show/solve

B_1 and ... and B_n and as logical implications: B_1 and ... and B_n implies H .

Experienced logic programmers use the procedural interpretation to write programs that are effective and efficient, and they use the declarative interpretation to help ensure that programs are correct.

This article attempts to set out the various similarities and differences between the various programming paradigms, and tend to serve some distinction between programming Languages that help to solve different problems that confront the human nature and those that do not.

3.3. Comparison to Imperative Programming

Functional programming is very different from imperative programming. The most significant differences stem from the fact that functional programming avoids side effects, which are used in imperative programming to implement state and I/O. Pure functional programming disallows side effects completely. Disallowing side effects provides for referential transparency, which makes it easier to verify, optimize, and parallelize programs, and easier to write automated tools to perform those tasks.

Higher-order functions are rarely used in older imperative programming. Where a traditional imperative program might use a loop to traverse a list, a functional program would use a different technique. It would use a higher-order function that takes as arguments a function and a list. The higher-order function would then apply the given function to each element of the given list and then return a new list with the results.

3.4. Criticism on OOP Language

A number of well-known researchers and programmers have analysed the utility of OOP, and have condemned the OOP in many ways. A sampled few of these are made here - Luca Cardelli states that it has bad Engineering properties[30], whilst Richard Stallman said that it is difficult to add OOP to Emacs because its not superior way to program.[31] Potok et al. Stated that there is no significant difference in productivity between OOP and procedural approaches[32]. Christopher J. Date stated that it is difficult to compare OOP to other technologies because of lack of an agreed-upon and rigorous definition of OOP. He and Darwen propose a theoretical foundation on OOP that uses OOP as a kind of customizable type system to support RDBMD. [33], and Alexander Stepanor suggested that OOp provides a mathematically -limited viewpoint and called it "almost as much of a hoax as Artificial Intelligence (AI), but was sorry to be unfair to AI[35] .

3.5. Differences in Programming Terminology

In summary, Imperative programming describes computation in terms of statements that change a program whilst Functional programming treats computation as the evaluation of mathematical functions and avoids state and mutable data. Procedural programming / structured programming specify the steps the program must take to reach the desired state and in Event-driven programming, the flow of the program is determined by events—i.e., sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads. Object oriented programming (OOP) uses "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs. Declarative programming expresses the logic of a computation without describing its control flow whilst in Automata-based programming the program or its part is thought of as a model of a finite state machine or any other formal automata.

None of the main programming paradigms have a pre-

cise, globally unanimous definition, let alone an official international standard. Nor is there any agreement on which paradigm constitutes the best approach to developing software. The above suggests that day-in-day out scientists and engineers are endeavouring to imitate something close to solving a problem just like the human brains do.

4. The Human Brains and How it works

According to Professor Jacob Palme of University of Stockholm, every animal you can think of -- mammals, birds, reptiles, fish, amphibians -- has a brain. But the human brain is unique. It gives us the power to think, plan, speak, imagine...etc. It is truly an amazing organ.

The brain performs an incredible number of tasks:

It controls body temperature, blood pressure, heart rate and breathing.

It accepts a flood of information about the world around us from our various senses (seeing, hearing, smelling, tasting, touching, etc).

It handles physical motion when walking, talking, standing or sitting.

It lets one thinks, dream, reason and experience emotions.

All of these tasks are coordinated, controlled and regulated by an organ that is about the size of a small head of cauliflower: your brain.

The brain, spinal cord and peripheral nerves make up a complex, integrated information-processing and control system. The scientific study of the brain and nervous system is called neuroscience or neurobiology.

In his article, Professor Jacob Palme explained that the brain works by activating thought modules called cognitive modules such as the following:

The modules controlling the hands when one rides a bicycle, to stop it crashing by minor left and right turns;

The modules which allows a basket-ball player to accurately send the ball into the basket;

The modules which recognized hunger and says that one needs food;

The modules which cause one to appreciate a beautiful flower, painting or person;

The modules which cause some humans to be jealous of their partners' friends;

The modules which computes the speeds of other vehicles and tells one if one has time to cross before the other car arrives;

The modules which tell one to look both to the right and to the left before crossing a street;

The modules which cause parents to love and take care of their children;

The sex drive modules; and

The fight or flight selection modules.

The above show that there exists several modules that work together to control the human body. However, the modules very important to this article are the those that control the thinking of a person.

4.1. Learned or Inherited

Professor Palme explained that the cognitive modules are learned or inherited, by saying that "Some of these modules are partly based on genetic inheritance, but also the inherited modules can be modified by learning. All one learns, in the childhood, and as an adult, will add new cognitive modules to one's brain. An adult human has millions of cognitive modules. The human species is unique in its capability to develop and modify cognitive modules by learning. Thus, the human species is successful because it is not so much controlled by instinct (genetic modules) and that it can modify or replace genetic modules by learned modules." The analysis drawn from the learned Professor shows that the human brains learns and inherits events in data processing.

4.2. Selecting the Right Cognitive Modules

How, then, does the human brain select the right module to apply to a certain issue? This Professor Palme explained by an analogy with a piano. A piano has a number of strings, one for each tone. If one lets a loudspeaker play a single tone loudly, then the corresponding piano string will begin to vibrate. Other piano strings corresponding to close matches, and to overtones of the played tone, will also start to vibrate, but to a less extent. All the piano strings receive the sound, but only those that match the sound will begin to vibrate. Thus, all piano strings test the sound at the same time.

In a similar way, when meeting a situation, this situation will simultaneously test many cognitive modules in the brain. To test many modules at the same time is known as "parallel processing" and is something which the brain is much better at than computers. But of all tested modules, only those which fit the situation best, are those which are most closely matched with the situation to be managed. The brain then has a selection mechanism, where the cognitive module which is most strongly activated takes over and is used as a model for how to handle the new situation. Examples of this selection mechanism are when one is feeling pain in different parts of the body at the same time, one is only conscious of the strongest of the pains. In the same way, lots of modules may react to one's situation, but only one or two of the strongest will make its way up to the conscious mind.

The human brain contains millions of billions of synaptic connections, in which the cognitive modules are stored. This vast size, and the capability to rapidly find appropriate modules in this large storage, is central to human intelligence.

4.3. Difference Between the Human Brain and Computers

It must be noted that the above description is very different from the way a normal computer functions. Mohammad and Krayem (Scrib Inc.) in defining expert systems de-

scribed that humans solve problems on the basis of a mixture of factual and heuristic knowledge. Conventional programs are based on factual knowledge, which are indisputable strength of computers. Heuristic knowledge composed of intuition, judgment, and logical inferences. Few computers have this facility of activating and matching millions of cognitive modules and selecting the appropriate one in a new situation. Especially the human capability to recognize cognitive modules which are in some way similar, but not identical, to a new situation, is unique for humans. Computers are good at finding identical situations, but not good at finding similar but not identical situations. This situation could be described of early computers that had single core processors.

In recent advancements in the developments of computer hardware, processor developers have made a break through in the development of multi-core processors. These have led to new developments in software in a form of threads-parts of a program that can execute independently of other parts- that can be executed in parallel mode.

In analyzing the various paradigms, it could be stated that each programming paradigm could solve a specific problem in live or business. However, it could be observed that natural problems and business problems are becoming complex and sophisticated and come with different combinations of solutions which need be coordinated to provide a holistic solution. With the modern computers providing multi-core processing capabilities its more a challenge to combine various programming paradigms to solve natural problems which hitherto had been difficult to be solved on the earlier single processor computers.

Let me sight a typical example of the development of Application software like Microsoft Word. The early word processors that were introduced in the early 1980's like Word Perfect, Word Star, ABC Writer, just to mention a few, had no dictionary nor Spell check. However, current Microsoft word could detect grammar, wrong spellings and even un-required spacing during typing a document, that shows that while one processor is listing the text being typed, another parallel processor will be checking spelling, grammar and spacing, showing that the Application has some level of cognitive aspects in its functions- a group of mental processes that includes attentions, memory, producing and understanding language, solving problems and making decisions. [41]

4.4. Audience

Today's businesses are in the competitive world, with the Top Executives dealing with large corporate data which are difficult to be handled. Businesses have gone out of creating extr ordinary databases, by creating Data Warehousing and using Data Mining tools to shift through volumes of data to discover hidden data attributes, trends and patterns within the Databases. But, these results do not provide holistic solutions to the business community and their Executives, because they have to sit down hours to analyze these trends, patterns and hidden attributes to determine the real

knowledge about the operations. With the breakthrough in the hardware developments, it could be possible for a holistic solution to be obtained through application developments.

5. The Situation Analysis – Programming Perspective

The diagram below Fig. 1 can demonstrate a best approach to current programming perspective.

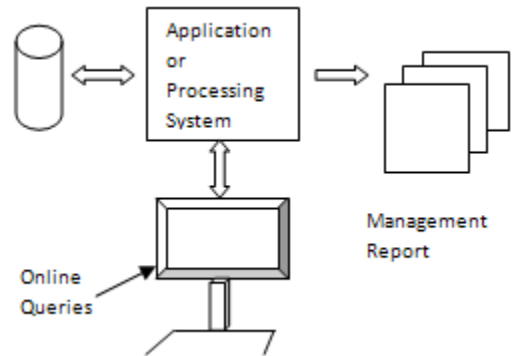


Fig. 1

The current business data processing are based on a number of independent created/ distributed databases which various applications process based on specific objectives to generate management reports. Many a time these reports are produced in hard copies or presented in graphical forms or tables for management to sit at long meetings to analyse the reports for decision making. The quality and accuracy in decision making from such reports may depend on the quality and technical knowhow of the management personnel present and their competence to critically analyse the obtained reports.

This leads to inaccuracies in management decision making resulting in absolute guesses and forecasts for the operations of the company or business, because it's difficult to compare current reports to earlier reports on the operations of the company some years back. Again, if these reports are obtained from different programming languages, analyzing these complex results will be hectic task. To avoid such situation and implement a holistic solution to business problems, the new approach of programming or developing Applications is to lower the risks involved in management making decisions based on guesses and forecasts.

5.1. The Cognitive Programming Paradigm of Next Generation Computing

The term 'cognitive' as borrowed from the functional behaviour of the human brain is used here to describe the next structure of computer programming, that indicates that the new generations of Applications would no longer produce summary information from a gathered database, but the summary information from various programming modules could be analysed and produce a holistic solution that

could come from combined solution of all modules to determine trends, patterns and hidden attributes that can logically formulate a final decision making process for scientists, medical Experts, Engineers and chief Executives to help them make right choice decisions. The flow chart Fig.2 depicts this concept.

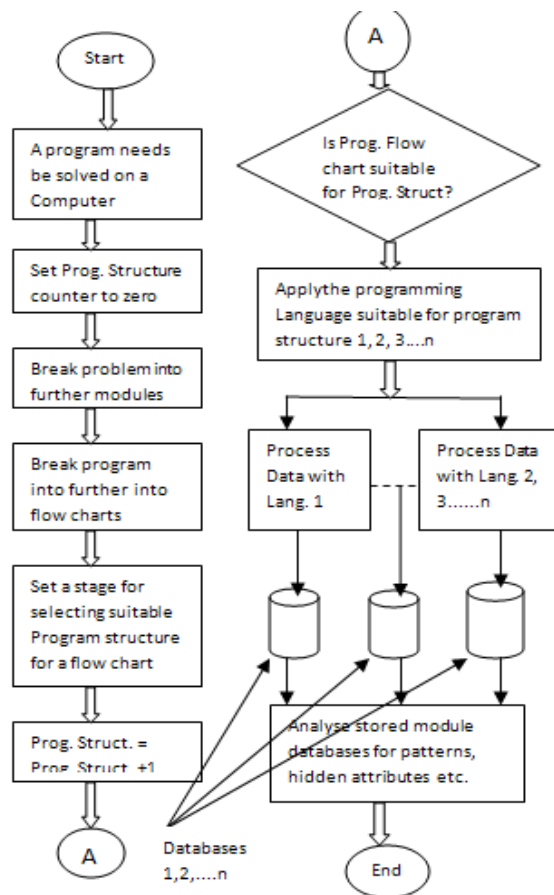


Fig. 2.

5.2. Design Concepts

The design concepts of the cognitive programming paradigm would include but not limited to the following:

computer should be broken into various modules;

Each module is further broken down to flow chart level;

Various programming languages depicting various programming paradigms are stored on computer hard disks;

A comparable analysis is done to select suitable programming language/program structure to a particular flow chart;

Problem modules are processed by suitable programming languages and structures and results are stored in various databases;

Adopt the human brains logic to compare various stored databases to determine trends, patterns, and hidden attributes produced by various programming paradigms logics of the new results with the past stored results to come out with alternative solution before providing final results; and

The final result of the last analysis would produce a definite information that could be the last and best result.

The above block schema depicts the various functional description of the cognitive programming Paradigm concept. With parallel processing and multi-core processors, the various modules' processes can be done in parallel to produce various data to be stored for final analysis to get the final results.

6. Conclusion

The cognitive programming paradigm as the author suggests to be the next programming structure, is to bridge the gap between the human thinking power with computer solutions in which databases are created for programming languages to search for data and analyzed them based on business logics used in designing the application. The other side effects of a problem's solution are not considered and these do not aid in arriving at a holistic solution. For example the human brains solution for a problem depends on the depth of knowledge and experience of the human being, and the problem's solution could combine a number of factors that will provide holistic solution, because the human brain processes a number of issues in parallel on a problem, and provides a holistic solution. Although the right or wrong of the solution depends on a number of factors available to the human being itself. With multi-core processors, programming structures defining different programming languages could be selected to solve various options of a problem in hand, so that analysing the various outputs holistic solution could be obtained.

References

- [1] Edsger Dijkstra, Notes on Structured Programming, pg. 6
- [2] Böhm, C. and Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules, CACM 9(5), 1966.
- [3] Michael A. Jackson, Principles of Program Design, Academic Press, London, 1975.
- [4] O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare Structured Programming, Academic Press, London, 1972 ISBN 0-12-200550-3
- [5] Edsger Dijkstra (March 1968). "Go To Statement Considered Harmful" (PDF). Communications of the ACM 11 (3): 147–148.
- [6] Jacobs, B. (2006-08-27). "Object Oriented Programming Oversold". Archived from the original on 2006-10-15. <http://web.archive.org/web/20061015181417/http://www.geocities.com/tablizer/oopbad.htm>.
- [7] Shelly, Asaf (2008-08-22). "Flaws of Object Oriented Modeling". Intel® Software Network. <http://software.intel.com/en-us/blogs/2008/08/22/flaws-of-object-oriented-modeling/>. Retrieved 2010-07-04.
- [8] Yegge, Steve (2006-03-30). "Execution in the Kingdom of

- Nouns". steve-yegge.blogspot.com. <http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html>. Retrieved 2010-07-03.
- [9] "The Jargon File v4.4.7: "syntactic sugar"". <http://www.retrologic.com/jargon/S/syntactic-sugar.html>.
- [10] "The True Cost of Calls". wordpress.com. 2008-12-30. <http://hbfs.wordpress.com/2008/12/30/the-true-cost-of-calls/>.
- [11] http://en.wikibooks.org/wiki/X86_Disassembly/Functions_and_Stack_Frames
- [12] Roberts, Eric S. (2008). "Art and Science of Java; Chapter 7: Objects and Memory". Stanford University. <http://www-cs-faculty.stanford.edu/~eroberts/books/ArtAndScienceOfJava/slides/07-ObjectsAndMemory.ppt>.
- [13] Roberts, Eric S. (2008). Art and Science of Java. Addison-Wesley. ISBN 978-0321486127. <http://www-cs-faculty.stanford.edu/~eroberts/books/ArtAndScienceOfJava/slides/07-ObjectsAndMemory.ppt>.
- [14] Guy Lewis Steele, Jr. "Debunking the 'Expensive Procedure Call' Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO". MIT AI Lab. AI Lab Memo AIM-443. October 1977. [1][2][3]
- [15] David Detlefs and Al Dosser and Benjamin Zorn (1994-06). "Memory Allocation Costs in Large C and C++ Programs; Page 532" (PDF). SOFTWARE—PRACTICE AND EXPERIENCE 24 (6): 527–542.)
- [16] Krishnan, Murali R. (1999-02). "Heap: Pleasures and pains". microsoft.com. <http://msdn.microsoft.com/en-us/library/ms810466%28v=MSDN.10%29.aspx>.
- [17] <http://microallocator.googlecode.com/svn/trunk/MicroAllocator.cpp>
- [18] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. University of Washington. doi:10.1.1.117.2420. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.117.2420&rep=rep1&type=pdf>.
- [19] Teaching FP to Freshmen, from Harper's blog about teaching introductory computer science.
- [20] M.Trofimov, OOP - The Third "O" Solution: Open OOP. First Class, OMG, 1993, Vol. 3, issue 3, p.14.
- [21] 21.Yegge, Steve (2006-03-30). "Execution in the Kingdom of Nouns". steve-yegge.blogspot.com. <http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html>. Retrieved 2010-07-03.
- [22] 22. Boronczyk, Timothy (2009-06-11). "What's Wrong with OOP". zaemis.blogspot.com. <http://zaemis.blogspot.com/2009/06/whats-wrong-with-oop.html>. Retrieved 2010-07-03.
- [23] Ambler, Scott (1998-01-01). "A Realistic Look at Object-Oriented Reuse". www.drdoobs.com. <http://www.drdoobs.com/184415594>. Retrieved 2010-07-04.
- [24] Shelly, Asaf (2008-08-22). "Flaws of Object Oriented Modeling". Intel® Software Network. <http://software.intel.com/en-us/blogs/2008/08/22/flaws-of-object-oriented-modeling/>. Retrieved 2010-07-04.
- [25] James, Justin (2007-10-01). "Multithreading is a verb not a noun". techrepublic.com. <http://blogs.techrepublic.com.com/programming-and-development/?p=518>. Retrieved 2010-07-04.
- [26] Shelly, Asaf (2008-08-22). "HOW TO: Multicore Programming (Multiprocessing) Visual C++ Class Design Guidelines, Member Functions". support.microsoft.com. <http://support.microsoft.com/?scid=kb%3Ben-us%3B558117>. Retrieved 2010-07-04.
- [27] Robert Harper (2011-04-17). "Some thoughts on teaching FP". Existential Type Blog. <http://existentialtype.wordpress.com/2011/04/17/some-advice-on-teaching-fp/>. Retrieved 2011-12-05.
- [28] Cardelli, Luca (1996). "Bad Engineering Properties of Object-Oriented Languages". ACM Comput. Surv. (ACM) 28 (4es): 150. doi:10.1145/242224.242415. ISSN 0360-0300. <http://lucacardelli.name/Papers/BadPropertiesOfOO.html>. Retrieved 2010-04-21.
- [29] Stallman, Richard (1995-01-16). "Mode inheritance, cloning, hooks & OOP". Google Groups Discussion. http://groups.google.com/group/comp.emacs.xemacs/browse_thread/thread/d0af257a2837640c/37f251537fafbb03?lnk=st&q=%22Richard+Stallman%22+oop&rnum=5&hl=en#37f251537fafbb03. Retrieved 2008-06-21.
- [30] Potok, Thomas; Mladen Vouk, Andy Rindos (1999). "Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment". Software – Practice and Experience 29 (10): 833–847. doi:10.1002/(SICI)1097-024X(199908)29:10<833::AID-SPE258>3.0.CO;2-P. <http://www.csm.ornl.gov/~v8q/Homepage/Papers%20Old/spetep-%20printable.pdf>. Retrieved 2010-04-21.
- [31] C. J. Date, Introduction to Database Systems, 6th-ed., Page 650
- [32] C. J. Date, Hugh Darwen. Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
- [33] Stepanov, Alexander. "STLport: An Interview with A. Stepanov". <http://www.stlport.org/resources/StepanovUSA.html>. Retrieved 2010-04-21.
- [34] Graham, Paul. "Why ARC isn't especially Object-Oriented.". PaulGraham.com. <http://www.paulgraham.com/noop.html>. Retrieved 13 November 2009.
- [35] Armstrong, Joe. In Coders at Work: Reflections on the Craft of Programming. Peter Seibel, ed. Codersatwork.com, Accessed 13 November 2009.
- [36] Mansfield, Richard. "Has OOP Failed?" 2005. Available at 4JS.com, Accessed 13 November 2009.
- [37] Mansfield, Richard. "OOP Is Much Better in Theory Than in Practice" 2005. Available at Devx.com Accessed 7 January 2010.
- [38] Stevey's Blog Rants
- [39] Rich Hickey, JVM Languages Summit 2009 keynote, Are

We There Yet? November 2009.

ing introductory computer science.

[40] Teaching FP to Freshmen, from Harper's blog about teach-

[41] [<http://en.wikipedia.org/wiki/cognition>].