Lorenzo Cioni

Co-operative principles in application design

(paper presented at AIHENP95, Pisa, Italy, 3-8 April 1995)

The main aim of the present paper is to explain some ideas about the use of co-operative principles for the composition of simple applications so to carry out complex tasks. In the introduction we set out our point of view, then we discuss some classic topics before presenting the basic and enriched data flow models and a brief discussion on some design tools.

1. Introduction

The basic idea from which we start relies on the empirical principle that the simpler an application is the more easily it can be designed, implemented and maintained. This principle becomes even more significant if we consider the software (sw) end-user as a sw developer or belonging to the community of sw developers. This perspective conflicts with that of commercial sw houses that tend to provide the users with very big and complex turnkey packages and has to face with the requirements of everyday problem solving. The need of solving more and more complex problems usually press towards applications of ever increasing complexity that require long periods of time to be designed and developed and can be very hard to modify and maintain. Moreover such applications entails the management of big projects with many people working together though divided in smaller groups, each group dealing with a sub-project until the lone programmer, that works on a particular aspect of the whole problem and produces a distinct piece of sw that must be assembled with many and many other so to build the final product. Though such topics play a fundamental role in Software Engineering, we get only a glance on them and focus our attention on some special simple technique for the design of complex applications starting from elementary independent modules[a]. Such simple techniques aim at bridging the evident gulf that exists between the need of quickly developing, modifying and maintaining complex application packages[b] and the effectively required times.

2. The program development cycle: some software engineering concepts.[1]

Software Engineering[2] (SE) essentially consists in the application of principles, skills and art to the design and implementation of programs and sets of programs. The starting point is the concept of computer system[c] that provides a set of services to its users. A computer system is the environment on which a program runs and a set of programs that carry out a desired set of functions and/or tasks represent a sw system. We can therefore define a hierarchy of linguistic levels as layers of sw, each one characterised by its own data types and by the primitive operations of its language. Each level represents the host system of the upper levels and the presence of such a hierarchy allows the independent design and implementation of both each level and of each component within that level.

The sketched structure is both the natural habitat of SE and the result of the application of its theoretical principles. SE's traditional end product is, indeed, the production of either sw packages or system sw. As to packages[d] they are designed according to a general model of their functionality and operation and of their users and are usually purchased on the basis of their supposed features. In many cases, however, they lack of some needed feature and this discovery can occur only during effective usage, since the use gives a better understanding of the problem under examination and a greater experience, so underlining restrictions and defects of the package.

In this case there are three possible solutions: accept the package as it is and redefine the complexity of the problem accordingly, buy a new and better package or throw it away and write from scratch a new package.[e] In the last case, the developer has to face with the program

---

[a] A module is a linguistic structure and depends, strictly speaking, on the abstraction level. We can consider, for the moment, a module as a block characterised by an input/output relation and a functional body.

[b] An application package is, at least, an application together with the on-line documentation and the proper portion of the file system.

[c] A computer system is composed by a hardware structure and a set of software elements.

[d] Similar considerations hold also for system sw , though they rarely originate within the users' community.

[e] Another solution could be either to use sw patches or to make changes to the source code, whenever available.

specification and with the choice of design, development, testing and maintenance methodologies. The first step tend to be the most important since an incomplete specification can produce unsatisfactory solutions. Following this path[f] we enter the program development cycle. The aim is the encapsulation of a strategy of solution into an algorithm and its translation into an executable form. We start with writing the specifications (we are interested in 'what' and not in 'how' and errors are very costly since turn into redundancy or wrong operation of some part of the code) then we pass to high level design (we define the skeleton), low level design (we flesh out the skeleton with details: the strategy is known so we define tactics), coding, build and test stage (where we polish the code so that its elements fit effortlessly well) to end with program maintenance and enhancement where run-time errors and deficiencies are corrected. To avoid too frequent feedback loops we could spend more time on specifications or build a prototype and keep more close and frequent contacts with the users but very often a program enters the enhancement phase since it needs to be either modified or extended so to adapt to the changing perception of the users. The process we briefly sketched obviously must fulfil some requirements that aims at the production of easy to use, expandable, reusable, compatible, efficient, portable and secure sw. Some compromises (for instance efficiency vs. reusability and integrity vs. performance) however cannot be avoided though can be handled somehow by the use of Object Oriented methodology.

3. Modularity and Object Oriented Programming (OOP).

In order to develop sw three approaches are available: top-down, bottom-up and middle-out. The first approach represents the computational counterpart of the "divide et impera" principle: an application is divided into simpler and simpler parts with minimal interactions and functional independence until the definition of modules that are primitive for the linguistic level at which we are programming. In the second case we compose primitive modules into more and more abstract ones until we obtain an application that perform a certain task and that we can encapsulate in a new module. In the latter case, we start from the definition of some "abstract" modules (for instance *play*, *record*, *display* and some others) then both compose them into a new application (for instance an application for DSP) and decompose them into smaller and simpler modules until we reach the primitive modules (for instance the statements of the chosen programming languages and the basic services of the Operating System). Modularity represents a concept that can be used in any of the aforesaid approaches. It is well suited to the bottom-up approach since it promotes both the reuse of existing sw and the development of models, theories and techniques for the design and implementation of reusable sw. Modules[3] are hence the elementary units of a program representation at a given linguistic level defined by a computer system. On its turn OOP represents a paradigm for the definition of classes of modules and their composition into larger modules by knowing only their interfaces and maintaining their integrity and their independence from the context. So to obtain sw that is adaptable, expandable and reusable one good choice is to found it on objects. OOP bases itself on modularity and each of its modules is designed to correspond with a well definite linguistic structure, to communicate with few other modules through explicit interfaces and, moreover, is characterised by private data, public data, accessible through the interface, and methods, the exported operations. OOP, therefore, allows the definition of applications starting from modules and the encapsulation of an application within a module so that such terms are synonyms.

4. The flow of data as a design tool.

Modularity represents therefore a stable concept in SE and the use of modules is common in application design. Though the design and implementation of simple modules can be usually carried out easily and effectively, once the proper domain for each has been chosen, their developer can be lacking in the ability both to manage their interactions and to compose them into useful and more complex applications. Tools for the efficient and effective development of application packages are therefore strongly needed. Such tools, besides allowing a good design of the flow of control, should also allow a control of the flow of data among the modules that are going to be composed to form a new application, even in the case where the modules correspond, at the implementation level, to processes rather than to subprograms and/or functions.

[f] We are well aware that our exposition is extremely concise. We hope it is clear enough. In what follows we disregard many topics and present as linear a process that, actually, is characterised by many feedback loops.

We propose therefore to analyse the flow of data among the modules so to set up relations (one-to-one, one-to-many, many-to-one) producer/consumer that allow, through the definition of the passing of data among modules, the design of complex applications.

For this purpose we can use the Data Flow (DF) model as a tool for the design, at the highest abstraction level, of complex applications starting from the analysis of the flows of data among the composing modules and to which there corresponds a virtual MIMD architecture that, on its turn, could be even embedded over a multitasking system, see paragraph 8.

5. The Data Flow model: some basic definitions

Within the basic DF model[4,5,6] a module M is characterised by a purely functional and deterministic input/output relation so that input values I are mapped over output values O through a generic characteristic function[g] F so that O=F(I). Input values are received on symmetric input channels and represent values produced by the activation of other modules. On their turn, output values are forwarded on asymmetric output channels and constitute input value for the activation of further modules. The behaviour of a module M is driven by the following *activation rule* (strict evaluation):[h] M waits for all of its input values, calculates F(I), produces its output values and sends them to those modules that have such values as elements of their input set. The granularity of the modules can be whatever from a single operation to a function of any complexity or even a full program whereas the natural structuring with modules can be used to develop highly parallel (and distributed) architecture.

The executable representation of a DF program is a directed graph where the nodes are the modules and the arcs are the communication channels through which the module exchange the data. The evolution of the computation can be represented by sketching the data on the arcs with tokens whose configuration defines the status of the computation. The aforesaid activation rule defines when a node is enabled: any node with a token on each input arc (and none on its output arcs[i]) is enabled and can fire so consuming input tokens and producing output tokens as a result of the evaluation of its characteristic function. The firing of the nodes defines a new status of the computation. A computation ends whenever there are no more tokens on the input nodes and the arcs are token-free. Arcs can carry either data or control tokens and can be used to express data dependencies among modules.

These features allow the independent execution of the modules so that the computation can evolve asynchronously at the natural speed of the modules, thanks to the absence of either synchronization or ordering mechanisms and of any size effect and logical dependencies. Another interesting feature is that the management of the I/O is primitive: the functional units are connected to peripheral devices and the communication among the drivers and their clients occurs through the exchange of tokens.

6. Composition rules and execution modes

Composition rules and execution modes come into play both as descriptive methods of the interactions among modules, as constraints on the design so to keep it plain and simple and as a way of approaching the implementation level.

Composition rules concern either the communications among modules or the access to persistent or volatile objects. As to the communication, such rules help us at defining one-to-one, one-to-many and many-to-one channels, client-server or partnership relations and concepts such as daemon node and group of nodes.[l] Partners evolve in parallel and are in a one-to-many or many-to-one relation with the same node whereas a group defines a sub-graph as a single unit, without hiding its inner structure.

As to the access to objects we introduce an extension in the notation though, from a theoretical perspective, we try to maintain the model clean. A permanent object is, indeed, either a file or a peripheral device such as the keyboard, the mouse, the screen or a dedicated

---

[g] We are aware of the formal nature of the mathematical concept of function and that in many practical cases subroutines and functions do not satisfy it but we would like to remember both that we are describing an abstract tool and that a looser definition that involves simply a relation among sets can be acceptable for our purposes.

[h] The input set I can be formed by the Cartesian product of a group of sets. The model allows also a loose evaluation according to which a module is activated when only the strictly needed data are present on its input channels: we ignore such a rule since, from our perspective , it is rather unreal.

[i] Such a constraint can be violated if we admit the forming of queues of tokens on the arcs.

[l] A group of nodes is a set of nodes that can be considered as a single node as opposed to the others.

hardware whereas a volatile object is essentially a portion of private or shared memory. From a conceptual point of view we define a daemon node as the manager of any of such objects.

Execution modes allow the definition of interactive nodes (bound to interactions with an external user), real-time nodes (bound to strict timings) and batch nodes (that supply their services at given times). Such nodes can execute either in the foreground (such as interactive nodes) or it the background (such as daemon nodes): they satisfy the trivial consideration that in any application only a small portion needs to be under the direct control of its user.

7. The enriched DF model

The DF model provides a good expressive power and the full explication of the independence among modules and so a potential parallelism.

We summarise here some extensions to the basic model so to enrich it as a design tool. The main classical extension concerns the need to express ways of handling resources and so the need to express history sensitive computations and requires the introduction of status tokens: the resource manager is a DF program that receiver requests from its clients and produces results for them and values of status for itself. From our perspective, the extensions we need are those to express within the graphs the aforesaid concepts. We, therefore, introduce graphic representations of objects, each with its daemon, of group of nodes, partners and execution modes (see paragraph 6).

8. Virtual MIMD: the first step towards the implementation.

At this point, we have sketched a tool for supporting sw development that is well suited for a bottom-up approach, favours sw reuse and exploit at the most both modularity and parallelism.

The proposed tool describes an application as corresponding to a graph of connected modules and resulting from the analysis of the flow of data within the application under construction. To step further, to each graph we can associate a data driven virtual MIMD so that each node corresponds to a virtual processor and the arcs are the communication channels among processors. We define so an intermediate abstract tool that is easy to use, is powerful and expressive, is hardware independent and can be easily mapped over a set of co-operating processes: each virtual processor can be mapped over one or more processes whereas FIFO streams or pipe can be used to implement the channels.

9. Conclusions

The present paper presents an abstract framework that is being elaborated and that refers to classical concepts to use them in a new perspective. Such a framework is a graphical way for supporting sw development and, thanks to an apparently easy correspondence with groups of processes, can be seen also as a way of describing sw while implementing it so that the two activities can be coincident. Much remains to do but the path seems promising. Among the things to do a formal language for the characterisation of the concepts introduced in paragraphs 6 and 7 and the description of the virtual MIMD and an environment for the implementation of the description with a set of co-operating processes.

References

1. D. Andrews and M. Greenhalg, *Computing for Non-Scientific Applications* (Leicester University Press, Leicester, 1987).
2. J. B. Dennis, in *Software Engineering. An advanced course*, eds. G. Goos and J. Hartmanis (Lecture Notes in Compter Science, Springer Verlag, Munich, 1973), p. 12.
3. J. B. Dennis, in *Software Engineering. An advanced course,* eds. G. Goos and J. Hartmanis (Lecture Notes in Compter Science, Springer Verlag, Munich, 1973), p. 128.
4. F. Baiardi, A. Tomasi , M. Vanneschi, *Architettura dei Sistemi di Elaborazione, Vol. 1,* (Franco Angeli, Milano, 1987).
5. P.C. Treleaven, in *Parallel Processing Systems. An advanced course*, ed. D. J. Evans (University Press, Cambridge, 1987, p. 275.
6. J. C. Syre, in *Parallel Processing Systems. An advanced course*, ed. D. J. Evans (University Press, Cambridge, 1987, p. 239.