Why UML will not do?

Introduction

The present stage of software development is in a large part dominated by an Object Oriented (OO) concept which influences the software development tools used. UML is a development tool supporting the OO concept.

The paper does not discuss critically some obvious flaws of UML (such as: its large and unwieldy nature, due to an attempt to fulfil too many wishes, difficulty of propagating any significant changes from code to its corresponding UML diagrams, overloaded and redundant graphical symbols, etc). They are discussed in detail in several publications, for instance in [2][3][4][6]. The paper concentrates only on two major flaws in the UML philosophy which are directly inherited from the OO concept.

Those flaws are decisive in the everyday life of software developers who by a lack of awareness or under pressure of the environment fall into the UML trap producing in consequence software which is more expensive and of worse quality than without UML.

In this paper we suggest that instead of trying to ignore some basic problems one should limit the use of UML to purposes for which it is suitable and not overuse it.

The true software challenges

Multitasking problem

Any non-trivial application requires multitasking software. Computer use multitasking for sharing a single CPU. Scheduling allows several programs to run apparently simultaneously giving the illusion of parallelism. The basic scheduling schemes are time-sharing and real-time systems with preemptive scheduling.

The solution offered by a PLC (Programmable Logic Controller) system (polling) as a replacement for multitasking works only for certain applications especially for systems which require a guaranteed, although not necessarily the fastest, response time. Most applications are better off with multitasking systems which provide not guaranteed but on average the fastest response time.

Control problem

Each program exhibits some control behaviour, understood as definite reactions to external stimuli. In many cases (for instance a control of industrial processes or aggregates) application software is built of several cooperating units. The design of such a complex control is not obvious and it is one of the greatest challenges for the present software development methods, tools as well as programmers involved in creating software systems.

Object Oriented software

The idea of OO software is based on a class concept. A class is a structure which contains both: data (member variables) and functions (member functions). The idea behind this concept is to expand the basic data types offered in a programming language (such as: bool, char, int, long, float in C++) with more complex types (classes) which allow encapsulation and hiding of data, the data being accessed and processed by member functions of the class. OO software is then constructed using objects of certain classes which cooperate with each other using their member functions. In this way

the methods of changing data are much more carefully controlled than if any software routine could alter data as global variables, for instance, and reliability is improved.

Weaknesses

The OO software concept does not offer solutions for communication between tasks. That is a completely foreign topic, outside the OOP concept. In other words the OO concept was not created to deal with that aspect of software.

The second challenge difficult to solve in an OO concept is the control flow. Software built according to the OO paradigm is a collection of objects which are instantiations of classes. It is relatively easy to include some control in the class as the class may contain functions which realize behaviour for the localised domain covered by the class. Unfortunately, a concept of how to realize behaviour of the entire application does not exist.

The realisation of control flow and the communication in multitasking system are the major challenges in any software project. The OO world does not provide solutions to those problems.

UML

UML (Unified Modelling Language) is a software specification tool. Essentially it is a graphical tool to support a specification of OO software. It has been intended to be a tool which allows the specification of the entire application software to be created and studied. The primary question is then what solutions are offered for the two problems which have any OO software: communication in multitasking system and realisation of a control task.

Communication in multitasking system

The answer is simple: UML does not offer any solution for this problem. It just assumes that the designer has to solve it in some way.

Realisation of control

UML recognizes the need to specify control flow. It offers Statecharts whose use is in practise limited to modelling behaviour in a class. The other element is the Sequence diagram which is used for specification of (more abstract) control which cannot be solved in the framework of a class. A link (or equivalence) between a Statechart and a Sequence diagram is not defined and left to the user’s interpretation or will. The third element is an Activity diagram which completes the two others but nobody understands why we need 3 incompatible formats for specifying control tasks. The Activity diagram is in fact a flowchart, and is intended as an extension to the Statecharts. The question never answered is why Statecharts requires this kind of extension.

The true problem is that while specifying software we have to express the behaviour which is the most error-prone part of the application. Therefore this aspect of the specification has to be of a piece, clear, accurate and readily understood. Its completeness determines the possibilities of testing and maintenance. Behaviour definition, scattered in different parts of the specification using different notations, is a nightmare and a source of never-ending discussions and confusion, leading to a failure to solve the essential problems of developing good software.

Several authors express their doubts about the alleged advantages of using UML, especially the alleged automatic code generation. A fairly representative picture emerges putting together the remarks summarized in Chapter 3 of [6].

Glass expressed his doubts [2] about “theoreticians' view of software modeling” citing among

others such a view:

"*... is clear that generating code from the evolving problem model is unlikely to work.*"

Mellor, a proponent of UML, finds only the following answer to that reproach, an answer which states that UML is just a text which a programmer translates into a code:

"*How comfortable are you with the idea that you could build a textual representation of a state machine and use it to choose which function to execute next in response to some signal?*"

In that context, in another paper [3] Douglas from iLogic (a company which sells Statecharts) admits:

"*You cannot translate Statecharts directly into a design-object model.*"

The author of a more recent paper [4] states:

"*The UML 1 version of activity modeling had a number of serious limitations in the types of flows that could be represented. Many of these were due to the fact that activities were overlaid on top of the basic state-machine formalism and were, therefore, constrained to the semantics of state machines*."

(Note: We do not agree that such constraints are either restrictive or harmful).

In spite of those comments and statements various UML marketing specialists "sell" it as a method for creating executable specifications which are automatically translated into code.

The future – UML2

We will be mistaken if we believe that UML2 will improve the situation. The following opinion found in [4] illustrates well the coming disaster: "*UML 2 replaced the state-machine underpinning with a much more general semantic foundation based on Petri nets, which eliminates all of these restrictions. In addition, inspired by a number of industry-standard business-processing formalisms, including notably BPEL4WS,21 a very rich set of new and highly refined modeling features were added to the basic formalism. These include the ability to represent interrupted activity flows, sophisticated forms of concurrency control, and diverse buffering schemes. The result is a very rich modeling toolset that can represent a wide variety of flow types.*"

The authors of the UML2 "discovered" that the difficulties are in parallelism. In a way that observation is correct: splitting the control among several objects we see the problem of dealing with several state machines (actually in a form of statecharts) running in several (in practice thread based) objects. However, that problem is voluntarily created, resulting from a wrong design approach. To believe that we "solve"it by using Petri nets is an illusion: designing control in a proper way is the correct answer to that challenge.

Petri nets are a nice concept for explaining parallelism. Difficulties that designers of control systems have to solve are not caused by parallel working of some control modules. A correctly designed control system must never depend on any parallel working modules. Let's explain this point using the well-known Dinning Philosopher problem [7] which illustrates the deadlock condition. The deadlock occurs if all philosophers pick up simultaneously one chopstick for instance on their left and wait for the chopstick on their right. This is a description of a wrong design and a Petri nets model shows the error in an understandable form. A technical solution of such parallel process must be done in such a way [7] that the problem never occurs: a philosopher (a process) grasps both chopsticks at the same time if they are available and the processes must be served sequentially.

If we use state machines organized as a hierarchical system the Slaves of any Master work in parallel but this does not have any significance for the Master. For the Master the Slaves' method (in parallel, semi-parallel or sequentially) of carrying out their task is irrelevant: the Master waits

until the Slaves reach required states to take the next control step. (This applies even if master and slave processes are running in different c.p.u.s or c.p.u. Cores). Therefore there are no convincing arguments to use Petri nets, invented for description of truly parallel problems, in UML. It would be a truly paranoiac situation if the Petri nets are intended for discovering errors in designs but in a way it would correspond to the not openly expressed but nevertheless governing paradigm of today's software development:

code with errors and then try to find them.

If you find this incredible, study the testing-based approaches suggested by the Agile programming community!

Conclusions

Overuse

People tend to overuse methods and tools. They try to exhaust all possible features which in extreme cases is contra-productive. There are many examples of this approach in software development.

The C programming language intended for the rather simple applications of the time, as a replacement for assembler programming, become *the* programming language in the ’80s of the last century. The result was: “*Hearing about a software catastrophe, we know it must have been written in C*”.

The concept of “exception”, correctly understood [1] as *a means to handle unexpected situations* (in other words: *if you have enough information to handle an error, it is not an exception*) becomes in some circles a solution for catching programming errors. The highlights of this completely erroneous interpretation of exception usage can be seen in the SCA (Software Communications Architecture) standard [5] which uses exceptions to signal incorrect function parameters. It means that all programming errors (as well as unexpected events) will be solved in exceptions. This idea is so ridiculous that it should not be taken seriously by experts in the field. It took us 25 years of discussions (especially in the C++ community) to decide about the sense and nonsense of using exceptions.

We have nowhere found any negative opinions about this obvious nonsense as manifested in the SCA standard: does the fact that it is a government-sponsored JTRS (Joint Tactical Radio System) standard frighten people from expressing their opinions?

A solution?

The standardization of modelling notations such as UML is unquestionably an important step for achieving better software specification. It is better to learn one notation instead of being confronted with ad hoc notations invented by anybody for each project anew. But the notation should be reasonable, it should not fail to keep its promises and we should not overuse it.

UML cannot overcome the obvious limitations inherited from the OO concept: the missing concept of multitasking systems and lack of- means for clear specification of a control flow. Therefore UML can be used for specification of sub-systems but cannot specify entire software systems. Still less can it be used to automate the later stages of detailed software production, e.g. coding.

Software considered as a weird and unstructured mixture of data processing and control will never be good software: it will be just a chaotic system full of ad-hoc local solutions which reflect the intellectual abilities of its authors. The use of UML in this process does not help, and indeed very

often results in slower and more expensive software development.

The real solution is to take another approach: modelling software as a system with strict partition between data and control flow and where the control flow supervises completely the data flow. The data flow is modelled according to OO concepts and the control flow is realized as a (preferably hierarchical) system of state machines. The practical implementation of course uses today's governing programming paradigm, i.e. the OO concept. Such solutions are already available as for instance StateWORKS [6].

References

[1] Eckel B., *Thinking in C++*, Vol. 2 (www.mindview.net/Books/TICPP/ThinkingInCPP2e.html).
[2] Glass R.L., "On Modeling and Discomfort", *IEEE Software*, March/April 2004, pp. 102-104.
[3] Douglas B.P., "UML for executable specification", *EDN*, August 2001.
[4] Selic B., "UML2: A model-driven development tool", *IBM Systems Journal*, Volume 45, Number 3, 2006.
[5] Software Communications Architecture Specification. JTRS-5000, SCA V3.0, 2004.
[6] Wagner F., Schmuki R., Wagner T., Wolstenholme P., Modeling Software with Finite State Machines: A Practical Approach. New York: Auerbach Publications, 2006.
[7] Wagner F., "Dining Philosophers", December 2003. www.stateworks.com/active/download/Dining-Philosophers.pdf.