ADA Language for Software Engineering

Diana ElRabih
Research & Development Department
Monty Holding Company
Beirut, Lebanon
E-mail: diana.elrabih@montyholding.com

***Abstract*— Software engineering is significantly more complex than just programming and as a result, then different tools are needed since software reliability cannot be compromised. ADA was designed as a coherent programming language for complex software systems, unlike other languages which grew by gradual addition of features. ADA is a modern programming language designed for large, long-lived applications and embedded systems in particular where reliability and efficiency are essential. Also ADA can be used as a communication language for some aspects of the needs and for some aspects of the design. In this paper, we present the concepts of ADA, as well as the strengths of ADA for software engineering.**

Keywords-ADA, software engineering, embedded systems, real time systems

I. INTRODUCTION

Software engineering is significantly more complex than just programming, and it should not be surprising that different tools are needed. The structure of the software market for personal computers has caused reliability to be consciously neglected. Software packages are compared by lists of features as performance (46 seconds is better than 47 seconds), and occasionally price. Vendors feel pressured to bring new versions to market, regardless of the reliability of the product. They can always promise to fix the bug in the next version. But word-processors, presentation graphics and interactive games are not the only type of software being developed. Computers are now controlling the most complex systems in the world: airplanes, spacecraft, communications networks, international banks, stock markets, military systems and medical equipment. The social and economic environment in which these systems are developed is totally different from that of packaged software. Each project pushes back the limits of engineering experience, so delays and cost overruns are usually inevitable. A company's reputation for engineering expertise and sound management is more important in winning a contract than a list of features. Consistent, up-to-date, technical competence is expected, not the one-time genius of a startup. Above all, system reliability cannot be compromised. The result of a bug is not just a demoted reporter or the loss of a sales commission. A bug in a medical system can mean loss of life. The crash of a communications system can disrupt an entire economy. The failure of a spacecraft can cost hundreds of millions of dollars. In fact, all these have occurred because of software faults. Software engineering is the term used to denote the set of techniques for developing large software projects. It includes for example, managerial techniques, such as cost estimation, documentation standards, configuration management and quality assurance procedures. It also includes notations and methodologies for analysis, design and testing of the software itself. There are many of us who believe that programming languages play an essential role in software engineering. In the end, a software system is successful if the 'code' of the program executes reliably and performs according to the system requirements.

The best managed project with a superb design is a failure if the delivered 'code' is no good. Thus, managerial techniques and design methodologies must be supplemented by the use of a programming language that supports reliable programming. The alternative to language support for reliability is 'bureaucracy'. The project manager must write conventions for interfaces and specifications of data representations, and each convention must be manually checked in code inspections. The result is that all the misunderstandings, to say nothing of cases where conventions were ignored by clever programmers, are discovered at best when the software components are integrated, and at worst after the software is delivered. Why cannot these conventions be formalized in the programming language and checked by the compiler? It is strange that software engineers, who make their living from automating systems in other disciplines, are often resistant to formalizing and automating the programming process itself.

ADA is a modern programming language designed for large, long-lived applications and embedded systems in particular where reliability and efficiency are essential. In fact, ADA is a design language as much as a programming language. It is designed to be read by ADA programmers and programmers not knowing ADA. Then from the point of view of the software engineers, in addition to being a programming language, ADA can be used as a communication language for some aspects of the needs and for some aspects of the design with its embodiment of modern software engineering principles. In this paper, we present the concepts of ADA, as well as the strengths of ADA for software engineering. ADA has rigid requirements for making entities such as subprograms and variables visible globally. This leads to a separation of ADA code into specifications or "specs" and bodies.

In section 2 we describe what ADA is, section 3 presents how ADA can be used for software engineering. In section 4 we talk about the development of ADA, while in section 5 we show the concepts of ADA. In section 6 we present the strengths of ADA. Section 7 concludes the paper.

 ISBN: 978-1-61208-668-2

II. WHAT IS ADA

The ADA language was designed to present a general language, unifying, standardized and supporting the precepts of software engineering. ADA is beginning to prove itself of reliability, robustness but has youthful defects. From 1990 to 1995 the revision of the standard leads to ADA95 , which corrects small defects, fills a big lack by making the language completely object (ADA is the first object language normalized). ADA95 adds its lot of novelties still unpublished 10 years later. Today ADA does not seem to have the place he deserves especially in first learnings of computer science where we must mix the programming itself with the good practice of programming. Since a long time ADA can largely replace Pascal the excellent teaching language. Paradoxically, ADA is more readily taught in high-level training because, again, it makes it possible to teach clearly, this time qualified and more arduous concepts. ADA is well used (even unavoidable) in avionics and embedded computing (rocket Ariane for example), as well as for traffic control (air, rail) where reliability is crucial. It is also appreciated when the code to develop is consequent (so very difficult to maintain). But the fact remains that currently few small or medium-sized companies admit to using ADA. Modest, productivity gains with ADA are proven and very significant.

Computer teachers eager to develop a quality code will be able to use ADA which is the culmination of procedural languages. A free, open and portable compiler (GNAT) allows (especially for academics) to run it and to adopt the language for a formation (or a culture) of computing.

Advantages of ADA by comparing to others: ADA appears more cost-effective compared to other similar languages [5]. ADA, unlike other languages which grew by gradual addition of features, was designed as a coherent programming language for complex software systems. In many instances in other similar languages to ADA such as C language, rules require a non-trivial amount of code development and verification, while the ADA solution is trivial [5]. For instance, achieving object initialization in similar languages requires the use of carefully implemented constructors, while specifying default initialization for ADA records is relatively trivial [5]. Another example is multi-threading with several rules for the use of locks, and condition variables. For ADA, the built-in facilities for direct task communication with protected objects for communication through shared buffers, includes implicit control of locks, and condition variables [5].

III. ADA FOR SOFTWARE ENGINEERING

The ADA language is complex because it is intended for developing complex systems, and its advantages are only apparent if engineers are designing and developing such a system. Then, and only then, they will have to face numerous dilemmas, and they will be grateful for the ADA constructs that help them resolve them. Next, we ask questions on ADA and we respond on these questions: How can I decompose the system? I can decompose the system into packages that can be flexibly structured using containment, hierarchical or client-server architectures. How can I specify interfaces? I can specify interfaces in a package specification that is separate from its implementation. How can I describe data? I can describe data with ADA's rich type system. How can I ensure independence of components of my system? I can ensure independence of components of my system by using private types to define abstract data types. How can data types relate to one another? Data types can relate to one another either by composition in records or by inheritance through type extension. How can I reuse software components from other projects? I can reuse software components by instantiating generic packages. How can I synchronize dozens of concurrent processes? I can synchronize dozens of concurrent processes synchronously or asynchronously. How can I get at the raw machine when I need to? I can get at the raw machine by using representation specifications. How can I make the most efficient use of my expensive testing facility? I can make the most efficient use of my experience testing facility by testing as much of the software as possible on a host machine using a validated compiler that accepts exactly the same standard language as the target machine.

Programming in ADA is not, of course, a substitute for the classical elements of software engineering. ADA is simply a better tool. The software engineers design their software by drawing diagrams of the package structure, and then each package becomes a unit of work. The effects caused by incompetent engineers, or by personnel turnover, can be localized. Many, if not most, careless mistakes are caught by type checking during compilation, not after the system is delivered. Code inspections can focus on the logical structure of the program, because the consistency of the conventions and interfaces is automatically checked by the compiler. Software integration is effortless, leaving them more time to concentrate on system integration. Though ADA was originally intended for critical military systems, it is now the language of choice for any critical system.

IV. DEVELOPMENT OF ADA

The ADA language was developed at the request of the US Department of Defense which was concerned by the proliferation of programming languages for mission-critical systems. Military systems were programmed in languages not commonly used in science, business and education, and dialects of these languages proliferated. Each project had to acquire and maintain a development environment and to train software engineers to support these systems through decades of deployment. Choosing a standard language would significantly simplify and reduce the cost of these logistical tasks. A survey of existing languages showed that none would be suitable, so it was decided to develop a new language based on an existing language, such as Pascal. There were several unique aspects of the development of ADA: The ADA language was developed to satisfy a formal set of requirements. This ensured that from the very beginning the

 ISBN: 978-1-61208-668-2

ADA language included the necessary features for its intended applications. The language proposal was published for scientific review before it was fully implemented and used in applications. Many mistakes in the design were corrected before they became entrenched by widespread use. The standard was finalized early in the history of the language, and facilities were established to validate compilers against the standard. Adherence to the standard is especially important for training, software reuse and host/target development and testing.

V. CONCEPTS OF ADA

This part is a quick overview of some of the strong points of the ADA language that allow engineers to discipline the development process and thus satisfy the precepts of software engineering.

A. Typing

ADA proposes few predefined types (so-called standard or primitive). Types: character, string, Boolean, integer numeric and actual numeric (comma floating). Their implementation is not specified by the standard and therefore it is recommended to define one's own, even the most basic, types in particular the numerals (integers, real floating and even real fixed) pledge of a safer programming (especially portability). ADA offers unparalleled power for the declaration of new types (and not necessarily numerical). This declaration inserts into the code a knowledge of the domain that usually stays in the comments, or in documents, or ... nowhere. This statement is portable and, in addition, the overflows are checked in the code generated. ADA is strongly typed which implicitly forbids mixtures.

B. Encapsulation

This property remains relevant with classes and objects. It is a question of rendering specific statement, closely intertwined the data of a software component and associated operations. Today, we are talking about member data and methods. In ADA, this software envelope is realized with the packages (package). But unlike Java (which takes this concept 20 years later). It is, in ADA, of a very concrete entity since declared in a clean file. This property is the pledge of a great federation of concepts where nothing is scattered. The package remained with ADA95 the basic structure of the language as it is a Robust and elegant code factorization technique.

C. Specifications and realization

The ADA package (ideal structure of a software component) houses these two entities well distinct (usually in two files). Package declarations on the one hand and package body on the other hand are the labels of these two entities (respectively spec and production). The spec part (not quite specifications but more surely a contract) only presents the declarations (data and sub-programs). The body of the package will realize, meanwhile, the contract proposed by the spec. Brand new entity (subroutine or package) relying on an already specified package announce this addiction with. The compiler then only refers to the part spec to control the syntax of the new entity. The coding of the realization can be deferred. This separation encourages prototyping without thinking about implementation and this development technique is very important. In the teaching of computing this process helps to force students to think before coding and it's the language (and the compiler) that gives teachers valuable help for this educational challenge. ADA provides novices with solid foundations that these will be able to transgress or use in other languages but with knowledge of be deferred. This separation encourages prototyping without thinking about implementation and this development technique is very important. In the teaching of computing this process helps to force students to think before coding and it's the language (and the compiler) that gives teachers valuable help for this educational challenge. ADA provides novices with solid foundations that these will be able to transgress or use in other languages but with knowledge of cause (and not out of ignorance). For example in the definition of subroutines the scope and direction of information exchanged is very clear. The passage of arguments for the procedures is specified in the prototyping. (Mentions In, Out, or In Out). Note that an ADA function remains a function (it accepts parameters in input and provides a single output result), for a novice, these basic notions appear very clearly. The impacts on the modification of the parameters during the use of an external procedure or function are equally clear.

D. Genericity

In ADA, these parameters (called of genericity) are as complex as desired. The parameters range from very traditional (constant or variable) through the types, subroutines and up to packages themselves generic! This profound degree of abstraction is absolutely remarkable. In ADA, the implementation of the genericity (declaration, instantiation and use) is very simple and elegant (so easy to teach). Packages generic ADA are compiled with authority without waiting for them to be instantiated which is not the case of C++ templates. This authoritative compilation validates the generic contract and to ensure that any instance respecting the contract will compile and will work as expected. Genericity allows and facilitates reuse and is even the safest technique to reuse reliably.

E. Exceptions

The exceptions are present in the language from ADA83. This concept is obviously essential in programming and allows to take into account the "anomalies" during the course of an application. ADA implementation of exceptions (declaration, initiation and treatment) seem to us of great ease and therefore very nice to teach. ADA95 has significantly improved this baggage and allows deeper treatments (a little less simple however to apply). In the same way, we point out a didactic property of importance namely the possibility of put in place, in the code, assertions.

 ISBN: 978-1-61208-668-2

F. Modular approach

This concept belongs to Object Oriented Design (OOD) techniques well illustrated with ADA83. A priori, when we stay at this stage of conception (and it is very often enough in many developments) it is not necessary to the "true" object. That is, it is often unnecessary to provide structures that to be extended by derivation (thus to make the design by analogy). However, if this is the case, ADA95 has an answer to make classes and objects.

G. Classes and objects

To make classes and therefore objects just take the concept of encapsulation, seen with the packages, and declare the first data structure (root) with the name tagged. Clearly any type 'tagged (or rather labeled) "is likely to be derived by extension and this inheritance characterizes the class structure. We can intelligently mix this design technique with the use of the hierarchies of packages seen above allowing, thus, even more flexibility and elegance in developments. Note, however, that the legacy multiple is not expected, indeed the designers of the language did not find useful to add a specific construct for multiple inheritance because too complex for a reduced use. On the other hand, conjunction "derivation and genericity" makes it possible to solve the cases of "mixing inheritance 'much less rare.

VI. STRENGHTS OF ADA

We are already talking about the software crisis and today, the problem is recurrent and even more worrying: development cost exceeded or difficult to predict, deadlines not respected, delivery not according to specifications, programs too greedy, unreliable, often impossible changes, etc. Let us review, briefly, some criteria or properties that must satisfy a consistent application and quality. We present in what follows the strengths of ADA language meeting the stated objectives.

a) Reusability: it is the ability of a software to be taken over, partly or even entirely, to develop new applications. We then talk about components software like the components that are reused in electronics.

b) Extensibility: it is the ease of adaptation of a software to the changes of specifications. Evolution of the data structure or adding features are desirable. This involves quick and reliable changes allowing a safe adaptive maintenance.

c) Portability: it is the possibility for a software product not to depend on a hardware environment, neither a system nor a particular compiler is particularly important for digital applications. Portability makes easier transfer of a hardware configuration and / or software system to another.

d) Testability: it is the implementation of aggressive processes whose purpose is to find errors in software. This phase of software development is important but often neglected or sloppy. This step is prepared before the implementation because we build test plans before coding (during the design stages).

e) Maintainability: it is the ability of a software to be modified elegantly, quickly, without fundamentally questioning the structure already specified or the existing applications.

f) Readability: it is the property of a code accessibility and understanding by more developers. The verbosity of a language sometimes decried can become a quality. As anecdote we could show the traditional "Hello World" to non-specialists, in different languages, The ADA version is the most readable (even compared to Pascal).

g) The ease of certification and validation: it is the ability of software to be able to be associated with properties proving that it meets its specifications, that it ends correctly or that it does not lock up in situations of load saturation or lack of resources. The existence of standardized language and verification possibilities helps to meet this goal.

VII. CONCLUSION

ADA is a design language as much as a programming language. ADA is designed to be read by ADA programmers and programmers not knowing ADA. From the point of view of the software engineers, in addition to being a programming language, ADA can be used as a communication language for some aspects of the needs and for some aspects of the design. With its embodiment of modern software engineering principles ADA is an excellent teaching language for both introductory and advanced computer science courses, and it has been the subject of significant university research especially in the area of real-time technologies. In our future work, we will plan to consider a case study in ADA showing an empirical study about advantages of ADA for software engineering.

REFERENCES

[1] M. Ben-Ari, ADA for Software Engineers, Weizmann Institute of Science, 2005.

[2] G. Booch and D. Bryan, Software Engineering with ADA, 3rd Edition, Addison-Wesley Professional, 1993.

[3] A. Wearing, Software Engineering, ADA and metrics, LNCS volume 603, 2005.

[4] D. Feneuille , “Teaching ADA-Choose a language: between the tendant and the reasonable”, Version 3,2, 2005.

[5] S. F. Zeigler , “Comapring Development Costs of C and ADA”, Rational Software Corporation, 1995.

 ISBN: 978-1-61208-668-2