Algerian Journal of Sciences-A, Special issue (NCAIIT'2021 conference) 150-158

REVUE ALGERIENNE DES SCIENCES
Section -A-
Sciences de la Nature et de la Vie
Sciences techniques
http://univ-eltarf.dz/fr/

AJPAN – Proposal of AspectJ Programming AssistaNtfor Learning and Teaching Introductory AOP Programming Concepts

Sassi BENTRAD [a]*

[a] LISCO Laboratory (Laboratoire d'Ingénierie des Systèmes COmplexes), University of Badji Mokhtar-Annaba, AlgeriaDepartment of Computer Science, University of Chadli Bendjedid El-Tarf, Algeria

Informations

Keywords:
Novice Programming
TeachingProgramming
Visual Programming
AOP
AspectJ
AJPAN

***Correspondence :**
sassi_bentrad@hotmail.fr
bentrad-sassi@univ-eltarf.dz

Abstract

Teaching and Learning Aspect-Oriented (AO) programming paradigm usually involves learning a programming language with a large amount of complexity. Novices very often spend more time dealing with syntactical complexity than learning the underlying principles of aspect-orientation or solving the problem.
Additionally, the textual nature of most programming environments works against the learning style of the majority of novices. Consequently, an academically programming tool for teaching and learning AOP is, therefore, desirable.
To this purpose, this paper offers a preview of the first stage in our research project, which we aim to evolve a new way for Aspect-Oriented Software Development (AOSD) paradigm and to develop an academic tool for both teachers and novices:**AJPAN – A**spect**JP**rogramming **A**ssista**N**t.

1. Introduction

The "Programming" or "coding" is a very useful skill and can be a rewarding career. In recent years, the demand for programmers and student interest in programming have grown rapidly, and introductory programming courses have become increasingly popular [1]. Learning to program is hard however. Novice programmers suffer from a wide range of difficulties and deficits. Programming courses are generally regarded as difficult, and often have the highest dropout rates. It is generally accepted that it takes about 10 years of experience to turn a novice into an expert programmer [1,2,13,26].

Programming is an abstract subject. It is believed to be hard to teach and to learn and many students in programming courses have difficulties to master all required competencies and skills. At the introductory level of programming, that problem is even more notable. The biggest learning problems are that the students have to handle concepts to which they do not have a concrete model in their everyday life, that they tend to approach programs line by line, and that they are not able to handle the larger wholes of the programs [7,12,13].

Generally in the programming phase, the adopted method depends on writing source-code by hand according to a specific language syntax. For a long time, programmers have done their work using tools that depend on the text-based style, and were often confronted with the difficulty of evolving their codes. During the understanding process, they may execute several tasks all together such as reading, searching, thinking, translating, recall and mental modeling, which make much harder the focus on specific problems [1,27].

Programming languages are the primary vehicles for supporting the practices of software engineering. To address various issues, it is important, therefore, that they should be well designed and implemented along with their supporting tools [10]. The latter must offer simultaneously a high-degree of flexibility and efficiency in code editors, so that making the programming process more efficient and also teaching and learning tasks [28]. There are many different attempts and a major effort has been directed to overcome this challenge. The modern editors come with some helpful features like *code outline, syntax coloring, highlighting and checking, code auto-completion*, and so on, in order to make the traditional programming style less

disheartening and boring, especially for novices having only basic understanding of concepts and programming-language constructs [2]. However, usually most of abstractions that are meaningful at the design may be lost when implemented at the coding phase [10].

In spite of these advances, using text-based editors still requires programmers to spend effort and focus on implementation details. A considerable research issues have been identified for making the act of "coding" relatively easy and effective, and researchers are focusing on bringing more improvements to the coding process [12]. In fact, numerous projects have investigated the ability of the Graphical User Interface (GUI) through other techniques like *Templates, Code-generators, Assistants*, and *Designers* in almost modern Integrated Development Environments (IDEs). In the last few years, the tendency of programming environments to support graphical techniques has been emphasized to provide the development, the execution and the visualization of the programs [11,32]. Unfortunately, most of them have proven success within limited domains as the case of *Visual Zero*[15], Tersus [41], etc. Most recently, a significant attention from the research community has been given for assisting the general-purpose programming tasks. The codeless program development represents one direction for prototyping and building quickly programs in a high-level of interactivity. It is a convenient way that tries to lessen the focus on formalisms by exploring the idea of code structure editor [40].

Aspect-Oriented Programming (AOP), a relatively new paradigm, recently earned the scientific community's attention. Nowadays, AOP is widely used in both academic and industrial world. Practice shows that AO-programs are in many cases shorter, have more modular structure and are easier to understand. Numerous publications discuss the advantages of AO-design and implementation. The complexity of an AO-program depends on the OOP components and the AOP specific constructs. Therefore the complexity could be scattered between the AO-specific parts (in pointcuts-definitions, advices, etc.), the OO-constructs (classes, inheritance, etc.), and even in the procedural-style implementation of the methods [27].

It has emerged initially at the programming level using strong implementations such as AspectJ, the de facto AOP standard language [25]. AspectJ encapsulates crosscutting concerns into new modular programming abstractions called "aspects" to preserve modularity instead of scattering them in the core modules "classes" [25]. However, programmers and especially novices experience some difficulties in using syntactic formalisms of some concepts and features [2,35].

In addition, the conventional coding is a tedious task that hinders their understanding, and often an impediment to effective programming. It can lead to repetitive stress due to the syntactic formalisms, what consequently, affects negatively the programmer's ability to be more creative. There is, therefore, a need for solid support tools to facilitate programmers' tasks. At the opposite, the codeless program development, which we advocate here, represents a way for building programs, with a significant decrease in the amount of code written, and less focus on detailed formalisms.

Learning AOP usually involves learning a programming language with a large amount of complexity. Students very often spend more time dealing with syntactical complexity than learning the underlying principles of aspect-orientation or solving the problem. Additionally, the textual nature of most programming environments works against the learning style of the majority of students. Consequently, an educational tool for teaching and learning AOP is, therefore, desirable. To this purpose, the aim of the first phase in our research project is to develop a programming assistant to support teaching of the AO-basic concepts with AspectJ, the most widely-used [26,27].

The remainder of this paper is organized as follows: Section 2 presents an overview of relevant background to the Aspect-Oriented Programming (AOP) paradigm, the learning and teaching of introductory programming concepts and the Visual Programming (VP) capabilities. Section 3 is devoted to introducing a preview of the first stage in our project, the overall architecture, followed by the process and technologies used in development. Finally, in Section 4, we summarize the conclusion from our preliminary work and future work avenue.

2. Background

2.1 Aspect-Oriented Programming (AOP) paradigm

Aspect-Oriented Programming (AOP) is a dynamic research field that focuses on the modular implementation of concerns (i.e., *non-business operations* such as: logging, authentication, threading, transactions...) that cut across a system's functionalities (i.e., *business logic*) [25].

It came to provide and to deliver a better separation of concerns (SoC). *Gregor Kiczales* coined the term as a complement to the object-oriented programming (OOP) paradigm rather than as a replacement to it [25]. However, there are still some problems, like implementation cost and its complexity were not in view, that defined by *Gail C. Murphy* as the main factors to keep in mind while

evaluating a software engineering methodology [14].

As with any new technology, AOP has both strengths and limitations in terms of their impact on software engineering. *Roger T. Alexander* and *JamesM. Bieman* reported a number of studies that explore these challenges [24]. *Muhammad Sarmad Ali et al.* [22] have performed a systematic literature review of empirical studies that explore the benefits and limitations of AOP-based development from the perspective of its effect on certain characteristics. According to their findings, a majority of the studies reported positive effects for code size, performance, modularity, and evolution related characteristics, and a few studies reported negative effects, where AOP appears to have performed poorly on cognitive dimensions of software development (i.e., cognitive burden issues) due to the new language constructs and mechanisms offered. Cognitive outcomes were measured by looking at two relevant factors: the time taken for understandability and development efficiency, which is measured in terms of the amount of time and effort spent to build programs. Obtained results were insignificant and not encouraging [23].
Although AOP is much more efficient and has been in existence for more than a decade, it has not gained the expected adoption as OOP, the most popular paradigm today. The reasons that have hindered its wide acceptability are: (1) the awareness (it is still less user friendly); (2) the lack of universal supporting framework; and (3) it has been still less heard of so technical experts are very few in number [24]. In addition, AOP introduced new dimensions and standards to programming. This, in general, creates complexity and possible resistance, but it was also the case when OOP was introduced, which indicates that this is a normal scenario [24].

Over the last few years, it has matured and received increasing attention from researchers across the world. Numerous works has been carried out on strong AOP-based implementations such as AspectJ language [25]. However, their acceptance in mainstream software development is still limited. They are mainly used only for maintaining, rather than for developing the initial version of a system. The prominent reason for this is the fact that support tools purely depend on the text-based style, which generally do not facilitate development tasks, as is the case of AJDT (AspectJ Development Tools [25, 42]) in spite of its completeness and maturity. A good overview of the AOP scene can be found at [44].

2.2 Teaching and learning programming concepts

A textual programming language can be learnt only through extensive training or education through books. One must have prior programming knowledge to advance his skills pertaining to textual languages. These languages provide facilities to express algorithms according to how computers operate but not according to how the human mind works. Since the medium of expression is text, which is one dimensional, algorithms are expected to be sequential. In addition, textual programming usually has a complex syntax, which is inherited from its natural language ancestry. Usually, the programmer is forced to follow strict rules to ensure that the program executes. This prohibits the programmer from being more creative and becomes more focus on the language syntax [28].

Programming allows novice students to explore creative topics and learn problem-solving skills. Programming learning is complex for many novices. The most important problem for many is their low ability to develop an algorithm that solves a given problem. The application of basic concepts or the design of simple algorithms can be difficult obstacles [2,5,9].

Teaching programming to novices has proven to be a challenge for both staff and students. Many students find the programming module difficult and disheartening in particular AOP and this could have an impact on their attitude to software development throughout the course and as a career choice. For staff involved in teaching programming it can also be very disheartening when students apparently fail to understand and be able to apply even the basic constructs [3,6,27]. These difficulties have prompted researchers to investigate tools and approaches that may ease the difficulty of teaching and learning programming [4,6]. Popular novice programming tools include *Verificator* [4], *ProfessorJ* [20], *RAPTOR* [37], *Alice* [38], *BlueJ* [39], etc. One of the most approaches that are being used for teaching programming is Visual Programming (VP) [1,3,4,31,33,34].

2.3 Visual Programming (VP) capabilities

AOP is the act of modelling systems in terms of aspects/objects-software descriptions of the behaviour of a part of the system. Researchers and developers are exploring how to combine VP with AOP to improve the ease of systems development, by investigating how the basic concepts of aspect and object-oriented programming create new opportunities for expressing systems in terms of visual construction.

How does one design a system in such a way that it allows novices (young programmers) to become experts without requiring the novice to abandon his or her hard won skills? The need for increased usability isn't something restricted to novices; even experts perceive the need for higher level

abstractions, leading to the development of graphics toolkits like *Open Inventor* [36].

The task of specializing programming environments for novices begins with the recognition that programming is a hard skill to learn. The lack of student programming skill even after a year of undergraduate studies was noted and measured in the early 80's and again in this decade [14]. We know that students have problems with looping constructs, conditionals, and assembling programs out of base components [8]— and there are probably other factors, and interactions between these factors, too. Not all of these potential environments have been built and explored, however. The field of Computer Science Education Research is too new, and there are too few people doing work in this field [5, 9]. According to *Nong Ye* and *Gavriel Salvendy* [10], technical experts have better knowledge of programming at an abstract level, and novices tend to have more concrete knowledge. Current research works seek to provide a higher-level of abstraction for developers by exploiting various graphical techniques during the development process [16,29]. They tend to make programming tasks easier for those having little background in the field, and may also be useful for the experienced ones for a fast software development or prototyping [32].

Visual Programming (VP) is a subject of current active research that has transformed the art of programming in recent years, aimed at reducing some of the difficulties involved in creating and using programs [16,11,19]. The main reason for using such techniques is that they are often more convenient to users than the traditional text-based style [33]. It allows representing the coding itself entirely or partially using graphical constructs instead of, or in addition to, the text-based coding [11]. In many cases, handling interactively visual representations offer significant advantages (for comprehension and development of large systems) over textual descriptions [12,29,30,31,34].

However, there is a common misunderstanding, which assumes that the research targets to eliminate the text-based method. In fact, this is a fallacy; most visual programming languages (VPLs) include text to some extent, in a multidimensional context. Their overall goal is to strive for improvements in the design of programming languages and associated tools. The opportunity to achieve this comes from the fact that in VP, we have fewer syntactic restrictions on the way a program can be expressed interactively, and this affords an independence to discover programming mechanisms that have not been possible formerly [11, 16, 17,30].On the other hand, unlike the text-based style that can be used for any coding tasks, the visual style is only suitable for certain tasks (limitation of suitability). For instance, in some cases of complex control structures like loops and recursion, the textual description is often more efficient and economic, and the code is usually more compact than visual programs.

3. AJPAN, an AspectJ Programming AssistaNt

3.1 Overview and Discussion

AOP provides new concepts that allow programmers to control the execution of programs acting on their control and data flows. A Responsible action on the two flows can implement a number of concerns ranging from the management of the competition to the persistence of the data; see the optimization of the calculations. Unfortunately, faced with these assets, the programmers facing difficulties to design and implement programs including concerns. In addition, the productivity is still restricted by a text-based primary input method at coding level, what makes program understanding, building, and maintaining more difficult.

Especially for novices without highly technical backgrounds, AOP complicated programming by combining two programming levels; for the low base code (core concerns: classes & interfaces) and domain-specific concepts (crosscuttingconcerns: abstract & concrete aspects). Our opinion on this issue is the use of a hybrid approach, a visual and text-based oriented method. It is a seamless integration between both to support novices' difficulties at coding time when combining these two levels.

The idea of codeless program development, such as under the tool *Limnor Studio* [40] whither it enhances the OOP by adding actions for reducing the hand-typing, is to make a general-purpose visual style that is simple and preserves the programming power. The trend was towards moving farther away from traditional editors through elevating the level of abstraction. The associated tools are changing the role of software engineering and allowing novice programmers to more easily developing and even getting quick overviews of large source-codes, which is difficult without higher abstractions. It is likely that this kind of tools will make the next-generation of development systems [10,21].

Significant advantages can be mentioned as follows:

- Developers can focus more on the design and innovations,
- Higher quality of code by avoiding potential and the most common programming mistakes, and
- Less time and effort to accomplish tasks efficiently, this boosts the productivity for huge systems.

The visual paradigm capabilities may not completely replace the conventional style of programming but it can enrich the textual view. The two forms can support each other in the educational context, development and maintenance activities. It is expected that the student will begin with the visual view, perhaps later moving on to the textual view as it allows them to perform some useful visual programs with a small investment of time and then go on to more advanced levels of understanding textually when they are ready [21].

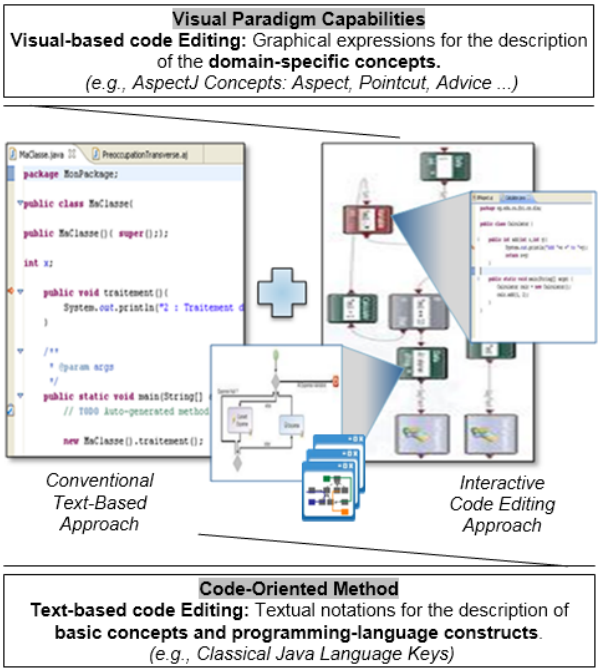


Figure 1. An Illustration of a Partially-Visual Code Editing.

An AO-programming language can be based on any programming approach—text-based, visual programming, or a combination of both. Incorporating a visual front-end to AspectJ language makes it more attractive and flexible. AO-programming concepts can be recognized and modified efficiently, and program structures can be parsed, understood, and manipulated more easily.

Our project is intended to better support the coding process by introducing more interactivity and a high-degree of flexibility. We seek to minimize the influence of language syntax on overall usability by using the ordinary drag-and-drop technique to overcome the weaknesses of text-based style.

We propose to take the AOP out of the conventional style of coding by using a new approach, which is partially visual. As the tool support is closely related to its base AspectJ language, mixing textual and visual programming is straightforward. We call it AJPAN. The name is an acronym for "AspectJ Programming AssistaNt", the language together with the tool are an attempt to integrate the visual and textual programming paradigms for one of the most AO-implementations applied to the Java language— AspectJ. The prototype tool presented here is currently being implemented as a set of plug-ins under Eclipse platform.

When displaying AO-concepts visually on screen, we can ideally combine the benefits of traditional score representations with the novel and dynamic possibilities of the computer. However, if the system becomes overly oriented toward visual programming, then having a visual front-end can become counterproductive, resulting in patches that are crowded and confusing. In these cases, the textual approach is often more economic and efficient. Thus, the user should ideally be able to switch to whatever programming technique is appropriate to a given problem.

3.2 Overall Architecture

The development of the **AJPAN** (**A**spect**JP**rogramming **A**ssista**N**t) tool begins with proposing the architecture, which consists of two parts, the structural architecture and the functional architecture. Both parts of architecture are discussed as follows.

Figure 2 depicts the overall structure of an open architecture of the tool. Open means that there are no limits for both internal and external extensibility. Basically it is implemented as a front-end to the AspectJ compiler (ajc). An AJPAN program is created on the tool editor by connecting a graphical objects or icons using a directed arrow that will produce a directed graph, which will show the flow of the program. This graph is also known as a source program (or an input file of data representation with the extension of .vaj). The front-end of the tool consists of three components, which are the syntax analyzer, semantic analyzer, and the code generator. The latter will initially produce an AspectJ source code (.java and .aj). Finally, an AspectJ code compiler used to produce a Java byte code (i.e. *.class), which is a target program.

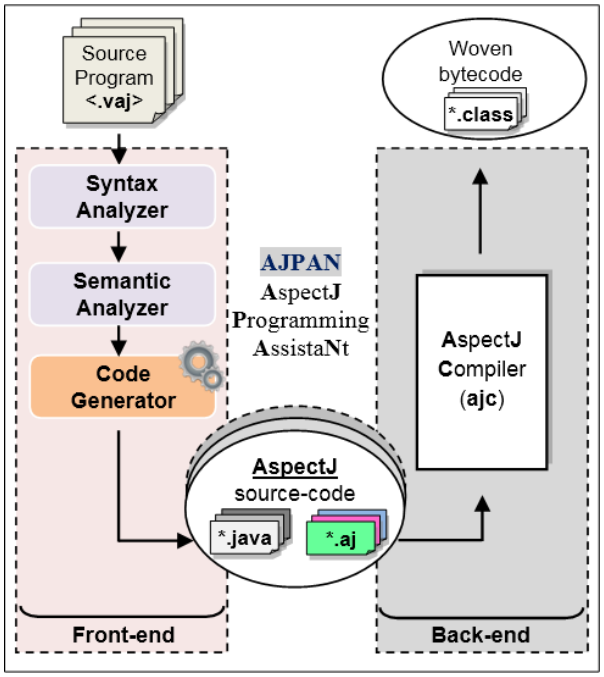


Figure 2. Structural Architecture of AJPAN tool support.

Figure 3 depicts an overview of the functional architecture of the AJPAN tool support prototype. It is an arrangement of functions and their sub-functions and interfaces, which define the steps sequencing for both control flow and data flow throughout the coding process. The facing down arrows represent the flow or steps of functionalities. The horizontal arrow shows the control flow, while the horizontal dash arrow shows the data flow.

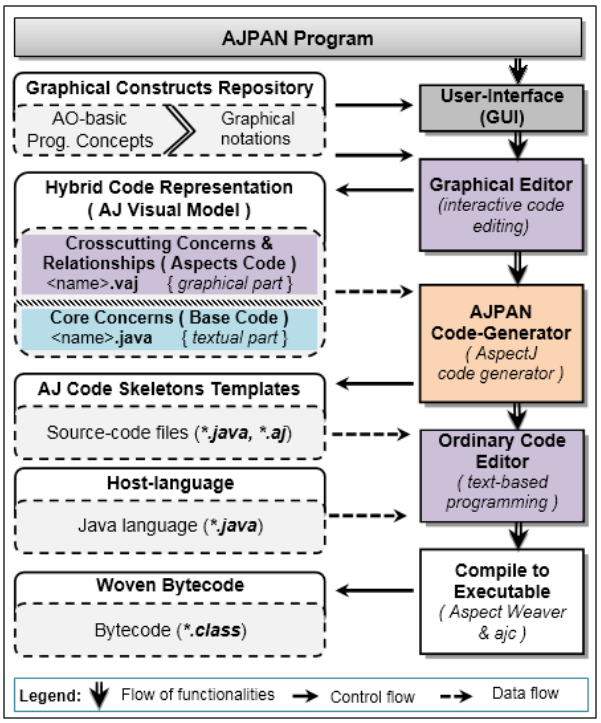


Figure 3. Functional Architecture of AJPAN tool support.

- **Interactive code editor**: is an innovative editor assistant, where the user can manipulate visual, interactive objects to build an*AJ Visual Model* (*.vaj) for code skeletons of the target program. It is a graphical editor of code structure providing a clear view that permits to obtain quick overview of concerns to be considered from the preliminary specification of the model. This tool is built as a novel class of visual, highly interactive code editor. Some of its important features include:
  - ✓ Less hand-typing of codes.
  - ✓ Exploring, navigating and modifying effortlessly the structure of program and its entities by means of its model, and without the need to look more deeply the source-code, and then regenerate corresponding templates.
  - ✓ Importing and exporting the already created models, what consequently allow reusing their constituent elements using a drag-and-drop technique.

- **AJPAN Code Generator**: this component is a template-based AspectJ code extractor based on Acceleo technology [43]. It takes as input the model already built, and produces as output textual templates of an AspectJ code.

- **AJ Visual Model**: describes the structure of the target program. In its simplest form, it seems as an interactive code visualization that may assist in understanding the overall program source. It consists of a set of *Model Elements* described in the form of graphical notations that represent AspectJ programming constructs and features. Each element can be customized, and has a specification according to the syntactic formalism description of the corresponding concept. The whole specification of the model data is stored as a relatively small file (*.vaj), defined in the XMI format.

  Furthermore, to elaborate additional details, the model can contain important artifacts such as a consistent documentation at different levels of abstraction (e.g., *UML models, graphs, tables, textual descriptions and comments, voice recording*, etc.). Having together such extra data in a single repository allows creating useful links between the generated code and these artifacts.

3.3 Design and Implementation

The AJPAN is an Eclipse-based prototype tool. In our approach processing, the compilation and execution of an AJPAN program is achieved in three phases.
First, the AJPAN program is a hybrid code, splited into graphical part (i.e. a set of *visual objects* or *V-*

expressions) and textual program and then replacing all the visual objects by unique identifiers. Additionally, a graphical constructs repository (predefined graphical notations & artifacts) indexed by these identifiers is created.

The second phase is the translation of each *visual term* (extracted from *V-expressions*) into its textual equivalent with a customized graphical parser.

In the last phase, the text-based produced code is parsed according to an attribute grammar for V+L and is then recombined with the textual equivalents of visual objects by means of syntax directed translation (an ordinary textual parser). The resulting program L-Program (i.e. a standard AspectJ program) is processed by a standard compiler for L (i.e. *AspectJ Compiler*).

The visual paradigm capabilities can facilitate programming tasks by using explicit and intuitive representations to express various aspects and entities of source-code [11]. To this end, it is desirable to provide carefully designed graphical notations for fundamental AO-basic programming constructs and features, such as aspect-constructs (*pointcut, advice, inter-type declarations, and weave-time declarations*), class-constructs (*method, field,* etc.), inheritance, and structured constructs (*package, class, interface, aspect, loops, conditions,* etc.) [21].

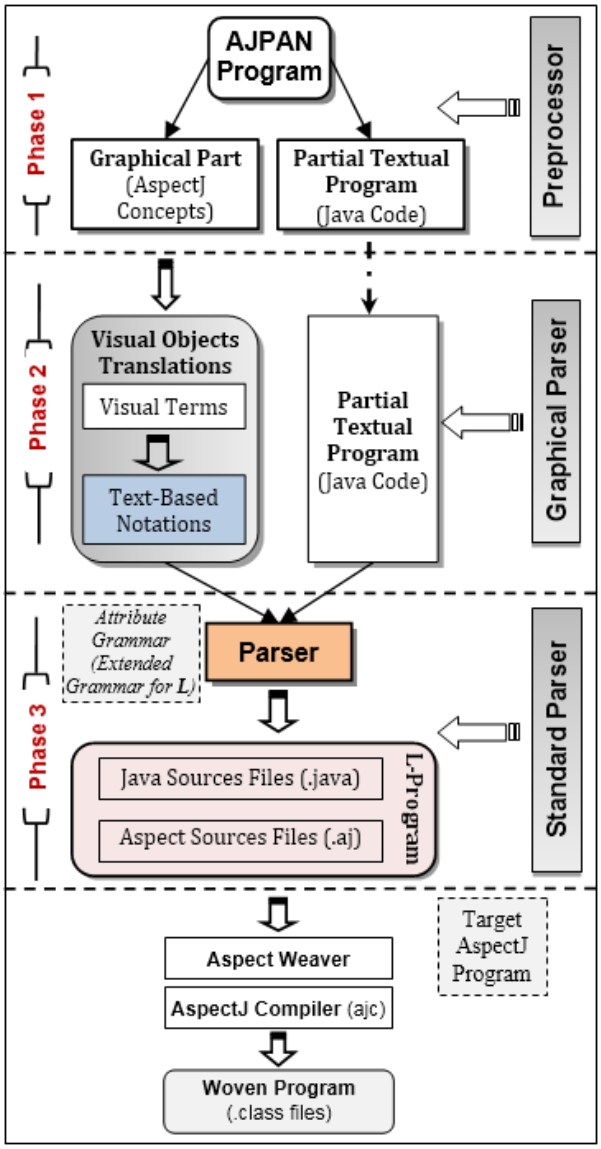


Figure 4. Overall process of Compilation and Execution of an AJPAN Program.

In more advanced implementation, it is good to let the program source take the form of a visual, interactive document [18]. In order to facilitate navigation, these representations must be coupled with efficient interaction techniques that permit to support two main functions: (1) controlling programming concepts through their corresponding constructs in a high-degree of flexibility; and (2) specifying parameters and properties for each model element selected. Based on this requirement, we have selected the following features to be implemented: (a) semi-separation among users and the language syntax; (b) no restrictions; and (c) an ordinary graphical user-interface. In this trend, the high-level descriptions encapsulate the underlying implementation technology adopted, which eases its replacement, e.g., replacing AspectJ implementation with AspectC++ [21].

4. Conclusion and Future Work

The AOP paradigm introduced new dimensions and standards to programming. Just reading about AOP concepts confuses the novices, let alone programming using these concepts. This, in general, creates complexity and possible resistance, but it was also the case when OOP was introduced, which indicates that this is a normal scenario.

For a long time developers have done their work using the text-based primary input method at coding level, but that is about to change. The success of new programming paradigm, such as the AOP, relies mainly on solid support tools and advanced development environment. We address the next steps towards successfully applying visual techniques to the software development process. Besides technical issues, one of the main concerns when defining a *partially-visual code editing* is finding a balance between the *expressiveness* and *implementability* of the tool support. Languages that are too expressive tend to be very difficult to formalize and therefore to implement. But by restricting the expressiveness of the language, we also restrict its usability. For that, we suggest to take the code-oriented method for describing the components, and the visual paradigm capabilities for expressing the structure and relations (i.e. the architecture) with a high-level of interactivity.

A new way for AOP and a preliminary work to develop, **AJPAN**– **A**spect**JP**rogramming **A**ssista**N**t, an academic programming assistant for AspectJ are presented and discussed in this proposal paper. In our current project, we suggest to use visual expressions for the description of the domain-specific concepts (e.g., *AspectJ Concepts: Aspect, Pointcut, Joinpoint, Advice* ...) in combination with textual notations for the basic concepts of the conventional (or standard) programming language (e.g., Java). Throughout its design, we are considering the aspect of understandability, easy to program, readability, writability and correctness of the language.

AJPAN will hopefully make programming less frustrating, more productive. We hope to be used for learning and teaching introductory AO-programming concepts. We suggest its adoption in the training levels of education as a goal of near-term, and to improve the quality and accessibility of information exchange between programmers and the computer, while at the same time supporting programs that solve large and complex problems as a goal of a long-term.

We believe that our programming assistant, AJPAN, will open a new-direction in the aspect-oriented software programming area. On the other hand, our proposal can be used to help researchers identify fruitful topics of future novice programming research, and towards a Hybrid methodology for the construction of AO software systems.

References

[1] Anthony Robins, Janet Rountree & Nathan Rountree: Learning and Teaching Programming: A Review and Discussion, Computer Science Education, 13:2, pp. 137-172, 2003.

[2] Lahtinen E., Ala-Mutka K., and Järvinen H.-M.: A study of the difficulties of novice programmers, ITiCSE 2005, Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, pp. I4-I8, June 2005.

[3] Klassen, M.: Visual approach for teaching programming concepts, In Proceedings of the 9th International Conference on Engineering Education, San Juan, Puerto Rico, pp. 23-28, 2006.

[4] Danijel R., Tihomir O., and Alen L.: Verificator: Educational Tool for Learning Programming, Informatics in Education Journal, Vol. 8, No. 2, pp. 261-280, 2009.

[5] Eckerdal Anna: Novice Programming Students' Learning of Concepts and Practise, Acta Universitatis Upsaliensis, Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 600, Uppsala, ISBN 978-91-554-7406-5, 2009.

[6] Robins A., Rountree J. and Rountree N.: Learning and teaching programming: A review and discussion, Computer Science Education, vol. I3, no. 2, pp. 137-172, 2003.

[7] Michael de Raadt: Teaching Programming Strategies Explicitly to Novice Programmers, PhD thesis, University of Southern Queensland, 2008.

[8] Fincher S. and Petre M.: Computer Science Education Research, Swets & Zeitlinger, 2004.

[9] Chin Soon Cheah: Factors Contributing to the Difficulties in Teaching and Learning of Computer Programming: A Literature Review. Contemporary Educational Technology 12:2, 2020.

[10] Nong Ye and Gavriel Salvendy, Expert-novice knowledge of computer programming at different levels of abstraction, Ergonomics, vol. 39, no. 3, pp. 461-481, 2007.

[11] Zhang K., Visual languages and applications, Springer-Verlag US, 2007.

[12] G. Lommerse, F. Nossin, L. Voinea, and A. Telea: The visual code navigator: an interactive toolset for source code investigation, in: IEEE Symposium on Information Visualization (INFOVIS), Minneapolis, MN, pp. 24-31, 2005.

[13] Ioannis V. Vasilopoulos, Paul van Schaik: Koios: Design, Development, and Evaluation of an Educational Visual Tool for Greek Novice Programmers, Journal of Educational Computing Research 57:5, pp. 1227-1259, 2019.

[14] G.C. Murphy, R.J. Walker, and E. L. A. Banlassad: Evaluating emerging software development technologies: lessons learned from assessing aspect-oriented programming, IEEE Transactions on Software Engineering, vol. 25, no. 4, pp. 438-455, 1999.

[15] Garcia Perez-Schofield J.B., Garcia Rosello E., Ortin Soler F., Perez Cota M.: Visual Zero: a persistent and interactive object-oriented programming environment, Journal of Visual Languages & Computing, vol. 19, no. 3, pp. 380-398, 2008.

[16] Ferruci F., Tortora G., and Vitello G.: Exploiting visual languages in software engineering, in: Chang S. K., Handbook of Software Engineering and Knowledge Engineering. River Edge, NJ: Singapore World Scientific, 2002.

[17] Aleksandr Miroliubov, Visual programming – an alternative way of developing software, Thesis Bachelor of Engineering: Information and Communications Technology, Metropolia University of Applied Sciences, 2018.

[18] French, G.W., Kennaway, J.R., and Day, A.M.: Programs as visual, interactive documents, Software: Practice and Experience, vol. 44, no. 8, pp. 911-930, 2014.

[19] Rémi Dehouck: The maturity of visual programming, 2015.

[20] GrayK. E. and Flatt: ProfessorJ: a gradual introduction to Java through language levels, In Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM Press, pp. 170-177.

[21] Bentrad S, Kahtan Khalaf H, Meslati D: Towards a Hybrid Approach to Build Aspect-Oriented Programs. IAENG International Journal of Computer Science (IJCS), Vol. 47, No. 4, pp. 677-691, December, 2020.

[22] Muhammad Sarmad Ali, Muhammad Ali Babar, Lianping Chen, Klaas-Jan Stol: A systematic review of comparative evidence of aspect-oriented programming, Information and Software Technology, vol.52, no.9, pp. 871-887, 2010.

[23] L. Madeyski and L. Szala: Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study, IET Software Journal, vol. 1, no. 5, pp. 180-187, 2007.

[24] Despi I. and Luca L., Aspect-oriented programming challenges, Anale Seria Informatica, vol. 2, no. 1, pp. 65-70, 2005.

[25] Ramnivas Laddad: AspectJ in action: practical aspect-oriented programming, Manning Publications Co., 2003.

[26] Bentrad S, Meslati D: A Way to Introduce and Reduce Difficulties of Aspect-Oriented Coding through AspectJ under Eclipse Platform, In The International Conference on Advanced Communication and Information Systems (ICACIS), Batna, Algeria, December 12-13, 2012.

[27] Bentrad S, Meslati D: Teaching and Learning Introductory Programming Concepts of AspectJ under Eclipse. In 1ières Journées Nationales sur l'Informatique et ses Applications (JNIAK), Khenchela, Algeria, April 29-30, 2012.

[28] Paolo Modesti: A Script-based Approach for Teaching and Assessing Android Application Development. ACM Transactions on Computing Education 21:1, pp. 1-24, 2021.

[29] Wen-Chin Hsu, Julie Gainsburg: Hybrid and Non-Hybrid Block-Based Programming Languages in an Introductory College Computer-Science Course. Journal of Educational Computing Research, 2021.

[30] Anne-Gaelle Colom, Wendy Purdy: Learn to Code, an Interactive Application to Promote Mobile Student-Centred Learning. Internet of Things, Infrastructures and Mobile Applications, pp. 232-241, 2021.

[31] Yue Hu, Cheng-Huan Chen, Chien-Yuan Su: Exploring the Effectiveness and Moderators of Block-Based Visual Programming on Student Learning: A Meta-Analysis. Journal of Educational Computing Research 58:8, pp. 1467-1493, 2021.

[32] Monika Mladenović, Žana Žanko, Marin Aglić Čuvić: The impact of using program visualization techniques on learning basic programming concepts at the K–12 level. Computer Applications in Engineering Education 29:1, pp. 145-159, 2021.

[33] Gary Cheng: Exploring factors influencing the acceptance of visual programming environment among boys and girls in primary schools. Computers in Human Behavior 92, pp. 361-372, 2019.

[34] João, Nuno, Fábio, Ana: A Cross-analysis of Block-based and Visual Programming Apps with Computer Science Student-Teachers. Education Sciences 9:3, pp. 181, 2019.

[35] F. Steimann: The paradoxical success of aspect-oriented programming, in: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), ACM, pp. 481-497, 2006.

[36] Open Inventor,http://www.vsg3d.com/open-inventor/more-than-a-3d-toolkit

[37] RAPTOR, http://raptor.martincarlisle.com/

[38] Alice,http://www.alice.org

[39] BlueJ, http://www.bluej.org

[40] Limnor Studio,http://www.limnor.com

[41] Tersus Project,http://www.tersus.com

[42] Eclipse AJDT Project,https://www.eclipse.org/ajdt

[43] Acceleo Project, http://www.eclipse.org/acceleo

[44] Aspect-Oriented Software Development (AOSD), http://aosd.net/