PAN4AJ: A Programming AssistaNt for Introductory AspectJ Programming *

Sassi BENTRAD *(a) and Djamel MESLATI *(b)

* LISCO Laboratory (Laboratoire d'Ingénierie des Systèmes COmplexes), UBMA, Algeria

(a) Department of Computer Science, University of Chadli Bendjedid El-Tarf , Algeria (ORCID: 0000-0002-7458-8121)
E-mail : sassi_bentrad@hotmail.fr , bentrad-sassi@univ-eltarf.dz

(b) Department of Computer Science, University of Badji Mokhtar-Annaba, Algeria
E-mail : meslati_djamel@yahoo.com

Abstract: Teaching and learning Aspect-Oriented Programming (AOP) usually involves learning a programming language with a large amount of complexity. Beginners 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 beginners. Consequently, a support tool for teaching and learning AOP is, therefore, desirable.

This paper offers a preview of the first stage of our research project, in which we aim to introduce a new way for the Aspect-Oriented Programming (AOP) paradigm. We intend to provide an easy-to-use educational tool for both teachers and beginners. We propose **PAN4AJ** – a **P**rogramming **A**ssista**N**t **for A**spect**J** language, the most widely used AO-based implementation. The language together with the tool are an attempt to integrate visual paradigm techniques with the text-based method to support programming activities.

Keywords: Programming assistant, Programming training, Academic programming tool, Teaching/Learning programming, Interactive Programming Environment, Educational Programming Environment, Visual programming; AOP/AspectJ.

1. Introduction

Programming is a very useful skill and can be a rewarding career. In recent years, the demand for programmers and students interested in programming has grown rapidly, and introductory programming courses have become increasingly popular [1,6,19]. However, programming has long been a common challenge in computer science classes across the world. Even in the early stages of programming introduction courses, it has a significant dropout and failure rate. It is a difficult topic that demands consistent work, a special approach, and multi-layer expertise. The process of acquiring various skills involves a lot of trial and error, as well as perseverance [5,27].

Programming is an abstract subject that is hard to teach and learn for many students in programming courses who have difficulty mastering all the required competencies and skills [7,9,27]. Programming courses are generally regarded as difficult, and often have the highest dropout rates [1,2]. At the introductory level, this problem is even more notable. The biggest learning problems are that students have to handle concepts to which they do not have a concrete model in their daily life, that they tend to approach programs line by line, and that they are unable to handle the larger parts of the programs [12,13].

Generally, the adopted method depends on hand-typing 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].

Programming languages are the primary vehicles for supporting the practices of software engineering. To address various issues, it is therefore important 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. Modern editors, such as Visual Studio Code, come with some helpful features such as code outline, syntax coloring, highlighting and checking, auto-completion, track changes, bookmarks and so on, in order to make the traditional programming style less disheartening and boring, especially for beginners who have only basic understanding of programming concepts and mechanisms [2]. However, usually most of the abstractions that are meaningful at the design may be lost when implemented at the coding phase [10].

According to *Nong Ye* and *Gavriel Salvendy* [10], technical experts have better knowledge of programming at the abstract level, and beginners tend to have more concrete knowledge. Current re-search 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 with little background in the field, and may also be useful for experienced ones for a fast software development or prototyping [32].

In spite of these advances, using text-based editors still requires programmers to spend effort and focus on implementation details. Considerable research issues have been identified to make 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).

Over the last few years, the tendency of programming environments to support graphical techniques has been emphasized to provide for the development, execution, and visualization of programs [11,32]. Unfortunately, most of them have proven success within limited domains, such as in the case of Visual Zero [15], Tersus [42], etc. Most recently, significant attention from the research community has been given for assisting the general-purpose programming tasks. The codeless pro-gram development represents one direction for prototyping and building quickly programs at a high-level of interactivity. It is a convenient way to lessen the focus on formalisms by exploring the idea of a code structure editor [41].

Aspect-Oriented Programming (AOP), a relatively new paradigm, earned the scientific community's attention to separate concerns and to build high quality software more easily, with higher maintainability, and a better chance at successful evolution [24]. 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 the paradigm at both the design and implementation levels.

It has initially emerged at the programming level using strong implementations such as AspectJ, the de facto AOP standard language [37]. However, programmers and especially beginners experience some difficulties in using syntactic formalisms of some concepts and features [2,36]. In addition, conventional coding is a tedious task that hinders the quick understanding of code and often is 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. Therefore, there is a need for solid support tools to facilitate the programmer’s tasks. At the opposite, the codeless pro-gram 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.

After being accepted by both a broad community of researchers and the industry, it is now being introduced in university courses. 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 tools works against the learning style of the majority of students. Becoming comfortable with some AO-programming basics first is an important step towards creating programs. Therefore, an educational tool for teaching and learn basic concepts is, therefore, desirable. To this purpose, the aim of the first phase in our project is to develop a programming assistant for teaching AspectJ, the most widely-used language [37].

The remainder of this paper is organized as follows:

Section 2 presents an overview of the relevant background to the learning and teaching of introductory programming concepts, the visual paradigm capabilities, and the Aspect-Oriented Programming (AOP) paradigm. Section 3 introduces and discusses the theoretical framework of our proposal. Section 4 is devoted to introducing a preview of the first stage in our proposed approach, the over-all architecture, the development process with the technologies and tools used followed by the pro-gram construction process. Finally, in Section 5, we summarize the conclusion from our preliminary work and the avenue of future works.

2. Literature Review

2.1 Teaching and Learning Programming Concepts

Programming is one of the most important aspects of a Computing course and is a challenging subject that requires a lot of effort from both teachers and students. It is a skill that several students find difficult to grasp when studying. Over the years, researchers have looked at the challenges of teaching and learning programming. The issue is more significant in computer courses, where learning programming is essential [19].

Teaching and learning programming are challenging tasks, especially for beginners, due to a number of factors, ranging from a lack of problem-solving skills to different teaching methods and support tools. It allows them to explore creative topics and learn skills to solve real problems [2,5,9]. Thus, it is axiomatic that to become an expert programmer, one needs extensive training coupled with a lot of practice and time. Furthermore, in addition to the methods and techniques used, dedication and constant hard work are also required for success [9,19].

One of the main issues related to teaching an introductory programming course is the excessive amount of time spent on the syntax of the text-based language, leaving little time to develop skills in program design and solution creativity [19,26]. Text-based languages rely heavily on the typing of commands. Usually, learners are forced to follow strict rules to form coherent programs and to ensure their execution. This can hinder them from being more creative, when focusing on the main aim of programming, that is, problem-solving, and become more focused on the language syntax [28].

Students, on their first time, find programming 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 the staff involved in teaching programming, it can also be very disheartening when students apparently do not understand and be able to apply even the basic programming concepts [1,3,9,27]. These difficulties have prompted researchers to investigate tools and approaches that may ease the difficulty of teaching and learning programming [1,4]. Therefore, it is necessary to have advanced tools that can assist in the teaching and learning process of initial programming. Many tools are now available. The most popular for beginners include Verificator [4], ProfessorJ [20], Raptor [47], Alice [48], BlueJ [44], Blockly [46], etc. One of the most popular approaches that is being used for teaching programming is Visual Programming (VP) [3,31,33,34].

2.2 Visual Paradigm Capabilities

AOP is the act of modelling systems in terms of aspects/objects-software descriptions of the behavior 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 OO/AO-basic concepts create new opportunities for expressing systems in terms of visual constructions.

The task of specializing programming tools for students begins with the recognition that programming is a hard skill to learn even after undergraduate studies. We know that beginners have problems with conditionals, looping, method structures, and assembling programs out of base components [8] –and there are probably other factors and interactions between these factors, too. However, not all of these potential tools have been built or explored. The field of Computer Science Education Research is too new, and there are too few people doing work in this field [5,9].

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 [11,16,19]. VP Languages can overcome the shortcomings of text-based languages. Common visual metaphors are introduced to represent statements and control structures as graphic blocks that can be composed to form programs, allowing programming without having to deal purely with textual syntax.

Program Visualization (PV) is an efficient and flexible way to inspect and analyze program data at several levels of detail during development and maintenance. Visualization techniques have long been used as an aid for both developers and engineers to quickly extract an overview of the hierarchical relationships between entities, to understand and manage the size and complexity of the source code [39,40].

The main reason for using such techniques is that they are often more convenient to users than the traditional text-based style [33]. A visual-based input method is better suited for beginners, as it improves the attitude toward programming [35]. 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 that assumes that the research aims to eliminate the text-based method. In fact, this is a fallacy; most VP languages 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 task, 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 the visual form, but this can only occur once the syntax and problem-solving processes have been mastered.

2.3 Aspect-Oriented Programming 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, etc.) that cut across a system's functionalities (i.e., business logic) [37].

As with any new technology, AOP has both strengths and limitations in terms of its impact on software engineering. *Roger T. Alexander* and *James M. Bieman* reported several studies that explore these challenges [24]. Muhammad Sarmad Ali et al. [22] have performed a systematic literature review of empirical studies that explore the advantages and limitations of AO-based development from the perspective of its effect on certain characteristics. According to their findings, most 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 for building programs. Obtained results were insignificant according to these two factors, especially for beginners [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) awareness (it is still less user friendly); (2) lack of universal supporting framework; and (3) it has been still less heard of, so technical experts are very few in number [25].

In addition, AOP introduced new dimension 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 [25].

Over the last few years, it has matured and received increasing attention from researchers. Numerous works have been carried out on strong AO-based implementations such as the AspectJ language [37]. 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 programming tasks, as is the case of AJDT (AspectJ Development Tools [37,43]) in spite of its completeness and maturity. A good overview of the AOP scene can be found at [45].

3. Theoretical Framework

In the following subsections, we introduce and discuss the theoretical framework for building an assistant tool for programmers (as visual and text-based support) from three perspectives:

a. The conceptual relations of visual paradigm capabilities, between Program Visualization (PV) and Visual Programming (VP) techniques,
b. A high-level description of a round-trip visual engineering system and
c. Its conceptual model articulating the interactions among the target program, the user-interface of an advanced code editor, and the user (a developer or a software engineer) within an integrated environment (IDE).

3.1 An Overview

Visual paradigm techniques may not completely replace the conventional style of programming, but they can enrich the text-based form. Both can support each other in the educational context, in development, and maintenance activities. Beginners are expected to start with interactive graphical views, and may later move on to the textual ones as it allows them to perform some useful programs with a small investment of time and then go on to more advanced levels of understanding textually when they are ready [21].

In addition, using various techniques of VP and PV conjointly within the same integrated environment (IDE) is highly desirable to improve the learning and teaching experience. The development of such environment is intended to help especially beginners to comprehend the multi forms of programs from a higher level to a lower level. The basic idea is to show programmers the programming stages, starting from designing problem-solving, constructing code, and validating the logical flow of the program through multilevel visualization of program abstraction. They can visually build their problem-solving skills. On the other hand, the system can simulate program behavior and data changes [38].

A simplified view of this idea, regarding the merging of these techniques for advanced software development, is shown in Figure 1 [39].

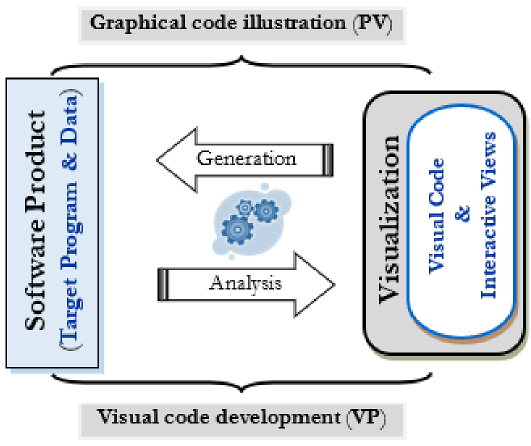


Figure 1. Combination of Visual Paradigm Techniques within an Integrated Environment.

Various visual paradigm techniques can be used at every step of the software development lifecycle. In addition, the efficient use of the human visual system through the visual paradigm can enhance both the understanding and productivity [38,39]. For the educational context, such contribution is intended to help students comprehend the multi forms of programs from a higher level to a lower level within an integrated environment.

3.2 A Round-Trip Visual Engineering (RTVE)

Actually, the community of software engineering has known a great deal of progress; novel approaches are emerging, as well as powerful languages and environments are now available. The Round-Trip Engineering (RTE) is considered to be the most efficient re-engineering approach in this progress through its several advantages such as the automation of the design process of a software product, the ability to derive implementations—decrease the effort at the implementation level without loss of manually created code—, and updating documentation automatically.

As shown in Figure 1, the visual paradigm techniques complement each other. PV generates visualizations from specifications of software products, while VP generates software products from visual specifications. Combining the two approaches together during the development process allows for a Round-Trip Visual Engineering (RTVE); for example, by producing visual presentations from the source code of an existing program code, changing these visual presentations, and generating a new program. This idea is included in our work and will be implemented within an integrated environment: toward a typical system for the engineering of huge and complex software products.

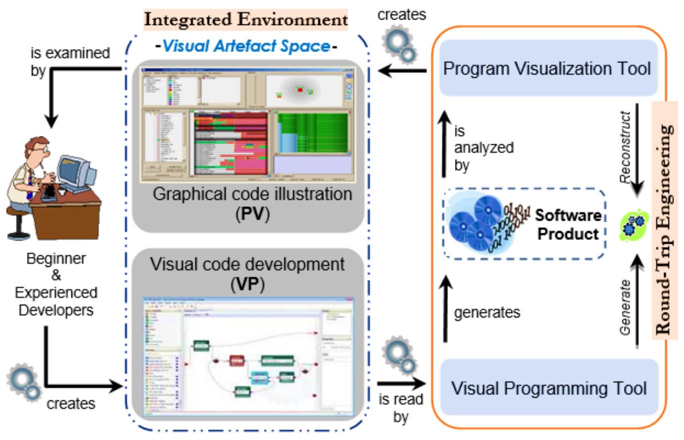


Figure 2. A High-Level Description of a Round-Trip Visual Engineering System.

A typical system should support the RTVE by incorporating both VP and PV where consistent graphical formalisms are desirable in order to maintain the user's mental map during the development process lifecycle. Figure 2 depicts an overall illustration of this round-trip style of engineering.

The system that we target to achieve through our proposed idea would technically be referred to as a VP tool, since the only qualifier in the PV definition eliminates any other possibility. That is, a VP tool can have some PV aspects, such as graphical debugging, and yet still be considered VP. However, it is unclear in our minds that such definitions were meant to apply to a system with extensive use of both PV and VP techniques, as would be needed in a reconfigurable system. Therefore, we anticipate the need for a new generation of tools that extensively incorporates both the PV and VP. Such an IDE would permit the graphical creation of programs from visualized preexisting source code. This code could be adjusted externally to the system, and subsequent system sessions would then reflect these changes using PV techniques. The code could then be configured interactively using VP techniques.

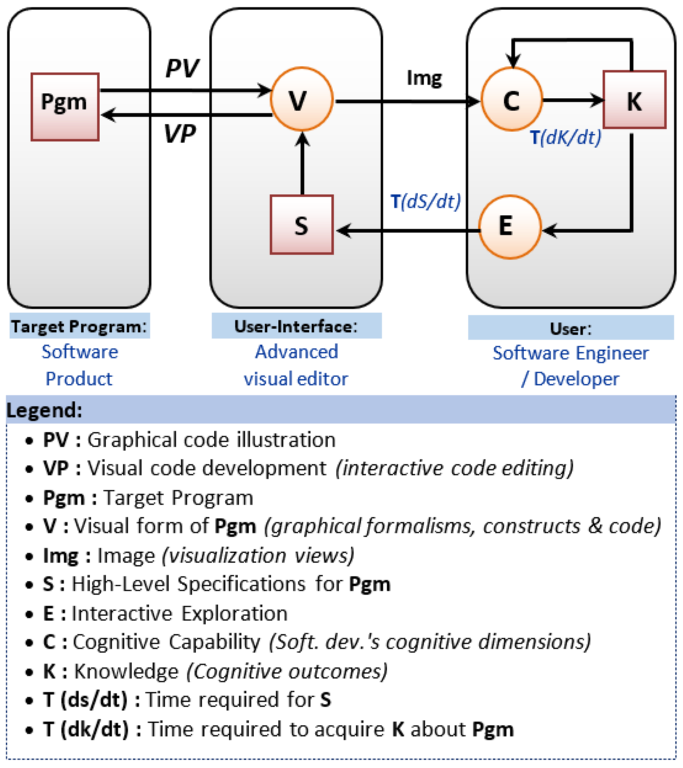
Besides the capabilities of visual modeling to be implemented that can help to model things easily, faster, and more accurately, the system can provide an intuitive navigation between code and visual model, Round-Trip model and source code synchronization.

3.3 The Conceptual Model

To explain the programming process supported by this new generation, a high-level conceptual model is needed to illustrate the roles of the User (developer/software engineer), the User-Interface (advanced visual editor), and the Target Program (software product).

We used the model of *Zhang* [11] —adapted model of the economic model of visualization proposed by *Van Wijk* that was used to identify the main ingredients as basic elements first, and then the associated costs and benefits are added [14]. This generic model is abstract and clearly defines the differences between the visual code development (VP) and the graphical code illustration (PV), which play complementary roles in the software development lifecycle.

Figure 3 illustrates our conceptual model. We consider the main ingredients of the User, User-Interface, and Target Program, rather than those of the User, Visualization, and Data in the context of visualization; the boxes denote containers; circles denote processes that transform inputs into outputs (i.e., in the basic model of *Van Wijk* [14]).


Pgm
PV
VP
V
Img
C
K
T(dK/dt)
S
T(dS/dt)
E
Target Program: Software Product
User-Interface: Advanced visual editor
User: Software Engineer / Developer

Legend:
- **PV :** Graphical code illustration
- **VP :** Visual code development *(interactive code editing)*
- **Pgm :** Target Program
- **V :** Visual form of **Pgm** *(graphical formalisms, constructs & code)*
- **Img :** Image *(visualization views)*
- **S :** High-Level Specifications for **Pgm**
- **E :** Interactive Exploration
- **C :** Cognitive Capability *(Soft. dev.'s cognitive dimensions)*
- **K :** Knowledge *(Cognitive outcomes)*
- **T (ds/dt) :** Time required for **S**
- **T (dk/dt) :** Time required to acquire **K** about **Pgm**

Figure 3. A Conceptual Model of a Round-Trip Visual Engineering System.

In the adapted model, the User is modeled with his or her Knowledge K about the Target Program Pgm to be developed, or to be analyzed and understood.

The Knowledge K is obtained through the Cognitive Capability C of the User, particularly the perceptual ability in the context of VP and PV. The Knowledge K also enhances Cognitive Capability C and plays the key role in driving the Interactive Exploration E through the User-Interface.

Through the interactive capabilities of the User-Interface, the User provides a high-level specifications S for the program Pgm to be constructed or the algorithms and their parameters to be applied. According to the specification provided, a program Pgm in visual form V is displayed and edited as an image Img.

In the context of Program Visualization (PV), the visual form V represents the graphical illustrations of a program's properties, such as status, structure, interaction among its components, or the output results. Its Image Img is perceived by the User, with an increase in Knowledge K as a result.

The target program Pgm is what the User is interested. It is to be constructed in the case of VP, or comprehended and analyzed in the case of PV.

In this conceptual model, the target program Pgm has a broader sense than the conventionally understood program. Pgm can be any of the following: (1) a code sequence conforming to a text-based programming language, (2) a code sequence conforming to a markup language which may not necessarily carry any computation (e.g., XML), or (3) a binary code or data structure generated from a high-level specification.

The visual code development (VP) refers to a process in which the User specifies the program Pgm in a multi-dimensional fashion (i.e., in the direction of V to Pgm). On the other hand, the graphical code illustration (PV) refers to a process in which certain properties of the program Pgm are displayed in a two or more-dimensional fashion according to the User's selection of parameters and/or algorithms. (i.e., in the direction of Pgm to V).

In general, visual paradigm techniques aim to simplify the process of program Specification S through graphical interaction and direct-manipulation techniques with minimal requirements of programming Knowledge K. The easiness of the Specification S for a given program Pgm is measured by the amount of Time T required, represented by ds/dt, while a PV tool targets at maximizing the User's gain in his or her Knowledge K about the program Pgm under analysis. The measurement of PV's effectiveness is made when the User takes Time T to gain additional Knowledge K(T)-K(0) about the program Pgm, represented by dk/dt.

4. Methodology

The following subsections will provide a preview of the first stage of our proposed method, as well as the overall architecture, the development process with the technologies and tools used followed by the program construction process.

4.1 PAN4AOP, Programmer AssistaNt for AOP

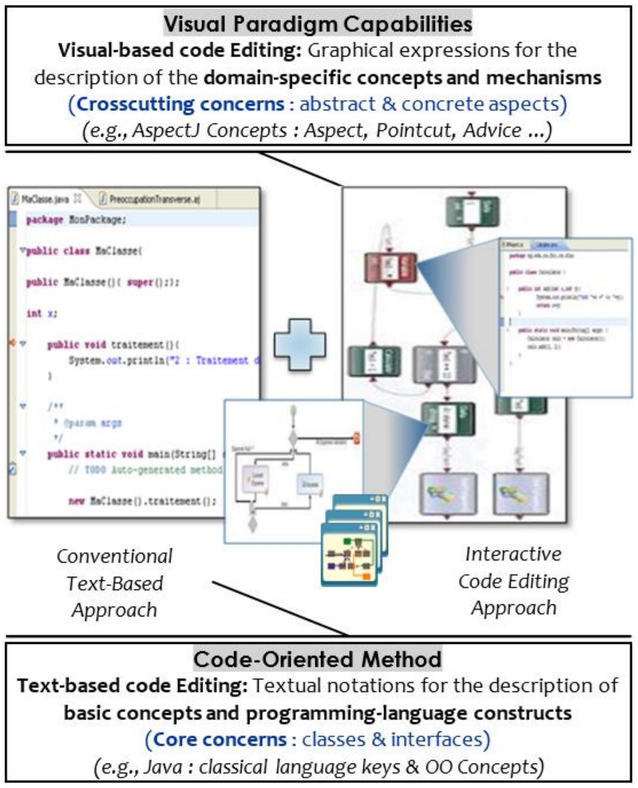
AOP provides new concepts that allow programmers to control the execution of programs acting on their control and data flows. A responsible action on the two flows can implement a number of concerns ranging from the management of the competition to the persistence of the data; see the optimization of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

Especially for beginners without highly technical backgrounds, AOP complicated programming by combining two levels; for the low base code (core concerns: classes and interfaces) and domain-specific concepts (crosscutting concerns: abstract and concrete aspects). The complexity of an AO-program depends on the OO-components and the AO-specific constructs. Therefore, the complexity could be scattered between the AO-specific parts and mechanisms (in pointcut-definition, advice, etc.), the OO-constructs (class, inheritance, etc.), and even in the procedural-style implementation of methods.

The software engineering community has recently been concentrating its efforts on moving away from traditional editors by increasing 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 challenging without higher abstractions.

As described in the conceptual model of the theoretical framework, combining the visual and text-based code editing (VP) and the graphical code illustration (PV) techniques within the same environment is highly desirable.

On this issue, we propose to use a hybrid approach, a visual and text-based oriented method, where some parts of the program are represented and manipulated graphically. It is a seamless integration between both to support beginners' difficulties at coding time when combining these two levels. A preview of a partially-visual editing of an AO-code (e.g., AspectJ program) is depicted in Figure 4.


Visual Paradigm Capabilities
Visual-based code Editing: Graphical expressions for the description of the domain-specific concepts and mechanisms
(Crosscutting concerns : abstract & concrete aspects)
(e.g., AspectJ Concepts : Aspect, Pointcut, Advice ...)
MaClasse.java
PreoccupationTransverse.aj
Conventional Text-Based Approach
Interactive Code Editing Approach
Code-Oriented Method
Text-based code Editing: Textual notations for the description of basic concepts and programming-language constructs
(Core concerns : classes & interfaces)
(e.g., Java : classical language keys & OO Concepts)

```
package MonPackage;

public class MaClasse{

public MaClasse(){ super();};

int x;

    public void traitement(){
        System.out.println("2 : Traitement d
    }

    /**
     * @param args
     */
    public static void main(String[]
        // TODO Auto-generated method

        new MaClasse().traitement();
```

Figure 4. An Illustration of a Partially-Visual Code Editing for AO-Programs.

The following are some of the most significant advantages:

a. Developers may focus more on design and creativity,
b. Code quality is improved by eliminating potential and common programming errors, and
c. Task completion takes less time and effort, resulting in increased productivity for large systems.

When expressing AO-concepts on the editor, developer can effectively combine the strengths of traditional score representations with the novel and dynamic possibilities of the computer. However, if the tool support 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 text-based editing is usually more economic and efficient. Consequently, the developer should ideally be able to switch to whatever programming method is appropriate for the solution and/or specification of a given problem.

This idea whither it enhances the coding process by adding interactive actions to reduce hand typing, is a first step towards making a general-purpose visual style that is simple and preserves the AOP power with an attractive manner.

4.2 PAN4AJ, First Tool Support Prototype

We show the practical feasibility and effectiveness of the approach through a prototype over Eclipse IDE, **PAN4AJ**. The name is an acronym for "**P**rogramming **A**ssista**N**t **for A**spect**J**"; the language together with the tool are an attempt to integrate the visual paradigm techniques with the text-based method to support programming activities. We have chosen AspectJ to be our first target AO-based implementation as it is the most frequently used and mature. AJDT is arguably the most complete, open and mature support, additionally regarded representative in the AOP research community [43].

4.2.1 Overall Architecture and Design

The development of the PAN4AJ tool support prototype begins with proposing the architecture with the main components and functionalities. Both parts, structural and functional, are discussed as follows. Furthermore, we discuss the architecture and design decisions that led to a decoupling of major functionalities and enable tool extensibility, interoperability, and end-user programmability.

- **Structural Architecture**

Figure 5 shows the overall structure of an open architecture. 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). The PAN4AJ program is created on the tool editor by connecting graphical objects using directed arrows that will produce a directed graph, which will show the program flow. This graph is also known as the source program (or an input file of data representation with the extension .vaj). The front-end of the tool consists of three components, which are a syntax analyzer, a semantic analyzer, and a code generator. The generator will initially produce an AspectJ code (.java and .aj). Finally, the AspectJ compiler is used to produce a Java byte code (i.e., *.class), which is the target program.

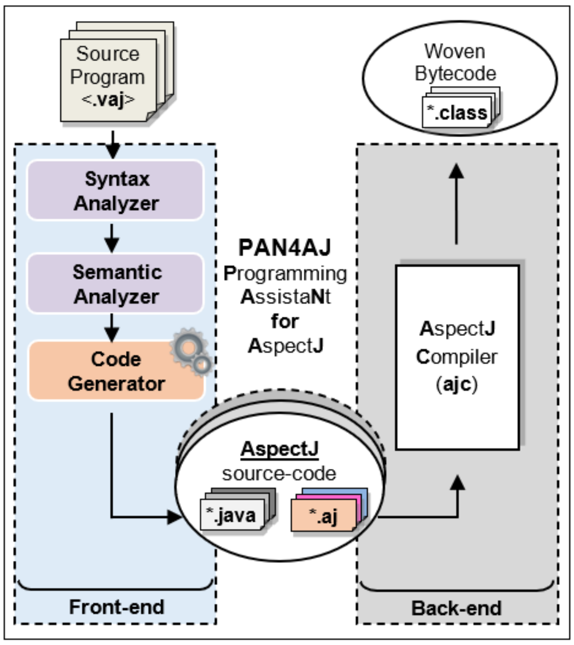


Figure 5. Structural Architecture of PAN4AJ tool support.

- **Functional Architecture**

Figure 6 shows an overview of the functional architecture of the PAN4AJ 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 arrows facing down represent the flow or steps of functionalities. The horizontal arrow shows the control flow, while the horizontal dash arrow shows the data flow.

  - **Interactive code editor:** is an innovative editor assistant, where the user can manipulate visual, interactive objects in order to build an AJ Visual Model (*.vaj) for code skeletons of the target program. It is a graphical editor of the 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 designed as a novel class of highly interactive code editor. Some of its important features include:

    a. Less code hand-typing.
    b. Exploring, navigating and modifying effortlessly the program structure and its entities by means of its model, without the need to look more deeply the source code, and then regenerating the corresponding templates.
    c. Importing and exporting models that have already been created, which consequently allow reusing their constituent elements using a drag-and-drop technique.

- **PAN4AJ Code Generator:** this component is a template-based AspectJ code extractor based on the Acceleo technology [50]. 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 a deeper understanding of 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 XMI format.

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 such extra data in a single repository allows creating useful links between the generated code and these artifacts.

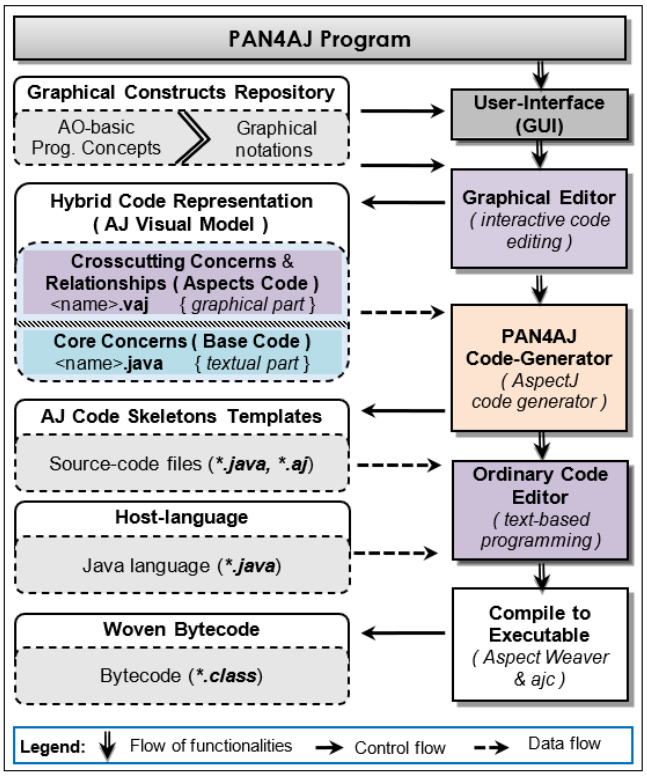

PAN4AJ Program
Graphical Constructs Repository
AO-basic Prog. Concepts
Graphical notations
User-Interface (GUI)
Hybrid Code Representation (AJ Visual Model)
Graphical Editor (interactive code editing)
Crosscutting Concerns & Relationships (Aspects Code)
<name>.vaj { graphical part }
Core Concerns (Base Code)
<name>.java { textual part }
PAN4AJ Code-Generator (AspectJ code generator)
AJ Code Skeletons Templates
Source-code files (*.java, *.aj)
Ordinary Code Editor (text-based programming)
Host-language
Java language (*.java)
Woven Bytecode
Bytecode (*.class)
Compile to Executable (Aspect Weaver & ajc)
Legend: Flow of functionalities
Control flow
Data flow

Figure 6. Functional Architecture of PAN4AJ tool support.

4.2.2 Technologies and Tools Adopted

The PAN4AJ is the first Eclipse-based prototype tool. As the programming assistant is closely related to its base language through the AJDT tool, mixing textual and visual programming is straightforward.

The underlying technology adopted is the Eclipse platform with its plug-ins capabilities. The approach exploits well known MDE (Model-Driven Engineering) technologies and tools provided under the Eclipse Modeling Project (EMP) [49,48], such as Eclipse EMF, GEF, GMF, and Acceleo technology [50], to enable building its graphical editor in addition to a code-generation engine.

To define a metamodel for a target AO-based implementation (e.g., AspectJ) we used EMF, while to build the code editor, we used GMF. We have chosen Acceleo as a Model-to-Text (M2T) transformation definition tool, mainly because of its very good support of EMF metamodels, to develop an automatic code-generation. Acceleo defines a template-based language used within an EMF-based tool support such as ObeoDesigner [50], for transforming models conforming to (i.e., an instance of) an EMF meta-model into text (source code).

4.2.3 User Interaction And Program Construction Process

More precisely, at the technical level, this first prototype is a general-purpose icon-based coding tool that completely leverages AJDT and includes initial implementation. It offers a hybrid-programming interface with the following features:

a. Within a visual editor, users can present the program's structure and certain behavior in a flexible combination of textual and graphical notations;

b. Visual representations are coupled with interaction techniques to simplify the navigation and understanding of the code. Users can easily verify the completeness of entities that make up the program source and the consistency of its relationships; and

c. Regardless of which development stage users are currently in, they can quickly check what they have developed. This is crucial in modern tools, where they have to deal with huge systems and are subject to information overload from various programming tasks: analysis, verification, testing, debugging, and so on.

As depicted in Figure 7, the compilation and execution of a PAN4AJ program is achieved in the following steps:

First, the PAN4AJ program is a hybrid code, splited into a visual part (i.e., a set of visual objects or visual-based expressions) and a text-based program and then replacing all graphical objects by unique identifiers. In addition, a repository of graphical constructs (predefined graphical notations and artifacts) indexed by these identifiers is created.

The second step involves using a customized graphical parser to translate each visual term (extracted from visual-based expressions) into its equivalent text-based notations.

In the last step, the produced code is parsed according to an attribute grammar (AG) for both visual-based (V) and text-based (L) parts (V+L) before being recombined with the textual equivalents of visual objects using syntax-directed translation (an ordinary textual parser). A standard compiler for L (i.e., AspectJ Compiler) is used to process the L-Program produced (i.e., a standard AspectJ program).

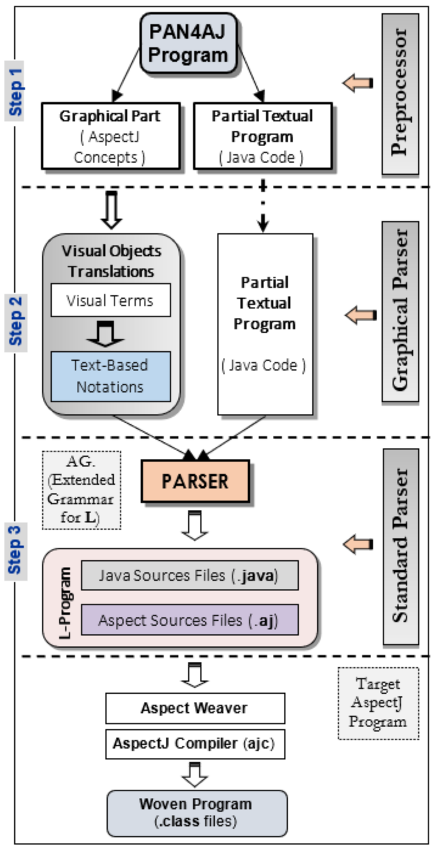

PAN4AJ Program
Step 1
Graphical Part (AspectJ Concepts)
Partial Textual Program (Java Code)
Preprocessor
Step 2
Visual Objects Translations
Visual Terms
Text-Based Notations
Partial Textual Program (Java Code)
Graphical Parser
Step 3
AG. (Extended Grammar for L)
PARSER
L-Program
Java Sources Files (.java)
Aspect Sources Files (.aj)
Standard Parser
Aspect Weaver
AspectJ Compiler (ajc)
Target AspectJ Program
Woven Program (.class files)

Figure 7. Overall process of the Compilation and Execution of a PAN4AJ Program.

The visual paradigm techniques 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 the 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].

In a 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:

a. Controlling programming concepts through their corresponding constructs with a high degree of flexibility; and

b. 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 between users and 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 facilities its replacement, e.g., replacing AspectJ implementation with AspectC++ [21].

5. Conclusion and Future Work

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

For a long time, developers have done their work using the text-based input method at the coding level, but that is about to change. The success of a new programming paradigm, such as the AOP, relies mainly on solid support tools under an advanced development environment.

In this paper, we address the next steps towards applying the visual paradigm techniques successfully in the development process. Our project is intended to take the current code-oriented method for describing the components, and the visual techniques for expressing the structure and relationships (i.e., the architecture) with a high level of interactivity. This idea whither it enhances the coding process by adding interactive actions with a high degree of flexibility to enhance teaching introductory AO-programming concepts, is a first step towards making a general-purpose visual style that is simple and preserves the AOP power with an attractive manner.

Becoming comfortable with some AO-programming basics first is an important step towards creating programs. Consequently, an educational tool for teaching and learning basic concepts is, therefore, desirable. **PAN4AJ** – **P**rogramming **A**ssista**N**t **for A**spect**J**, is our first programming assistant for teaching and learning AspectJ. The language together with the tool are an attempt to integrate the visual techniques with the text-based method to support programming activities for one of the most widely-used AO-based implementations.

We believe that this programming assistant will open a new direction in the aspect-oriented software programming area. On the other hand, this 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.

We conclude by offering our direction for future practice and research:

First, besides technical issues, one of the main concerns when defining 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. However, by restricting the expressiveness of the language, we also restrict its usability.

Second, the next stage of our work will be the experimental confirmation of our proposal: carrying out usability testing and user acceptance tests. We could test the assistant internally with a few participants to gather insights and then use the feedback to inform the design decisions.

Third, related to generalizing our proposal according to the theoretical framework, further research into the issue would be of interest. The proposal can be replicated by using the same methodology with other AO-based implementations such as AspectC++, etc.

References

[1] Robins, A., Rountree, J. and Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2), 137–172. doi: 10.1076/csed.13.2.137.14200

[2] Lahtinen, E., Ala-Mutka, K. and Järvinen, H. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin*, 37(3), 14–18. doi: 10.1145/1151954.1067453

[3] Klassen, M. (2006). Visual approach for teaching programming concepts, *9th International Conference on Engineering Education*, San Juan, Puerto Rico, 23–28.

[4] Radosevic, D., Orehovacki, T. and Lovrencic, A. (2009). Verificator: Educational Tool for Learning Programming. *Informatics In Education*, 8(2), 261–280. doi: 10.15388/infedu.2009.16

[5] Bosse, Y. and Gerosa, M. A. (2017). Why is programming so difficult to learn?: patterns of difficulties related to programming learning mid-stage. *ACM SIGSOFT Software Engineering Notes*, 41(6), 1–6. doi: 10.1145/3011286.3011301

[6] Robins, A. (2019). *Novice programmers and introductory programming*. The Cambridge Handbook of Computing Education Research, Cambridge Handbooks in Psychology, 327–376.

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

[8] Henriksen P. and Kölling M. (2004). Greenfoot: combining object visualization with interaction. *ACM SIGPLAN Conf. Object-Oriented Program. Syst., Lang.*, Appl., 73–82.

[9] Cheah, C. (2020). Factors Contributing to the Difficulties in Teaching and Learning of Computer Programming: A Literature Review. *Contemporary Educational Technology*, 12(2), ep272. doi: 10.30935/cedtech/8247

[10] Ye, N. and Salvendy, G. (2007). Expert-novice knowledge of computer programming at different levels of abstraction. *Ergonomics*, 39(3), 461–481. doi: 10.1080/00140139608964475

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

[12] Lommerse, G., Nossin, F., Voinea, L. and Telea, A. (2005). The visual code navigator: An interactive toolset for source code investigation. *IEEE Symposium on Information Visualization (INFOVIS)*, 24–31. doi: 10.1109/infvis.2005.1532125

[13] Vasilopoulos, I. and van Schaik, P. (2018). Koios: Design, Development, and Evaluation of an Educational Visual Tool for Greek Novice Programmers. *Journal Of Educational Computing Research*, 57(5), 1227–1259.

[14] Van Wijk, J. (2006). Views on Visualization. *IEEE Transactions On Visualization And Computer Graphics*, 12(4), 421–432. doi: 10.1109/tvcg.2006.80

[15] García Perez-Schofield, J., García Roselló, E., Ortín Soler, F. and Pérez Cota, M. (2008). Visual Zero: A persistent and interactive object-oriented programming environment. *Journal Of Visual Languages & Computing*, 19(3), 380–398. doi: 10.1016/j.jvlc.2007.11.002

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

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

[18] French, G., Kennaway, J. and Day, A. (2013). Programs as visual, interactive documents. *Software: Practice And Experience*, 44(8), 911–930. doi: 10.1002/spe.2182

[19] J. Figueiredo and F. García-Peñalvo. (2021). Teaching and Learning Tools for Introductory Programming in University Courses. *International Symposium on Computers in Education (SIIE)*, 1–6. doi: 10.1109/SIIE53363.2021.9583623.

[20] Gray, K. E. and Flatt, M. (2003). ProfessorJ: a gradual introduction to Java through language levels. *18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, 170–177.

[21] Bentrad S., Kahtan Khalaf H. and Meslati D. (2020). Towards a Hybrid Approach to Build Aspect-Oriented Programs. *IAENG International Journal of Computer Science (IJCS)*, 47(4), 677–691.

[22] Ali, M., Ali Babar, M., Chen, L. and Stol, K. (2010). A systematic review of comparative evidence of aspect-oriented programming. *Information And Software Technology*, 52(9), 871–887. doi: 10.1016/j.infsof.2010.05.003

[23] Madeyski, L. and Szala, L. (2007). Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study. *IET Software*, 1(5), 180–187. doi: 10.1049/iet-sen:20060071.

[24] Alexander, R. T., Bieman, J. M. and Andrews, A. A. (2004). Towards the systematic testing of aspect-oriented programs. *Technical Report CS-4-105*, Colorado State University, Fort Collins, Colorado.

[25] Despi I. and Luca L. (2005). Aspect-oriented programming challenges, *Anale Seria Informatica*, 2(1), 65–70.

[26] Samer A., Javanshir A. and Mohamed A.N. (2006). On the Development of a Programming Teaching Tool: The Effect of Teaching by Templates on the Learning Process. *Journal of Information Technology Education: Research*, 5(1), 271–283.

[27] Kadar, R., Wahab, N. A., Othman, J., Shamsuddin, M. and Mahlan, S. B. (2021). A Study of Difficulties in Teaching and Learning Programming: A Systematic Literature Review. *International Journal of Academic Research in Progressive Education and Development*, 10(3), 591–605.

[28] Modesti, P. (2021). A Script-based Approach for Teaching and Assessing Android Application Development. *ACM Transactions On Computing Education*, 21(1), 1–24. doi: 10.1145/3427593.

[29] Hsu, W. and Gainsburg, J. (2021). Hybrid and Non-Hybrid Block-Based Programming Languages in an Introductory College Computer-Science Course. *Journal Of Educational Computing Research*, 59(5), 817–843. doi: 10.1177/0735633120985108

[30] Colom, AG. and Purdy, W. (2021). Learn to Code, an Interactive Application to Promote Mobile Student-Centred Learning. In: Auer, M.E., Tsiatsos, T. (eds) *Internet of Things, Infrastructures and Mobile Applications*. IMCL 2019. *Advances in Intelligent Systems and Computing*, vol 1192. Springer, Cham.

[31] Hu, Y., Chen, C.-H. and Su, C.-Y. (2021). Exploring the Effectiveness and Moderators of Block-Based Visual Programming on Student Learning : A Meta-Analysis. *Journal Of Educational Computing Research*, 58(8), 1467–1493. https://doi.org/10.1177/0735633120945935

[32] Mladenović, M., Žanko, Ž. and Aglić Čuvić, M. (2021). The impact of using program visualization techniques on learning basic programming concepts at the K–12 level. *Computer Applications In Engineering Education*, 29(1), 145–159.

[33] Cheng, G. (2019). Exploring factors influencing the acceptance of visual programming environment among boys and girls in primary schools. *Computers in Human Behavior*, 92, 361–372. doi: 10.1016/j.chb.2018.11.043

[34] João, Nuno, Fábio and Ana. (2019). A Cross-analysis of Block-based and Visual Programming Apps with Computer Science Student-Teachers. *Education Sciences*, 9(3), 181. https://doi.org/10.3390/educsci9030181

[35] Saito, D., Washizaki, H. and Fukazawa, Y. (2017). Comparison of text-based and visual-based programming input methods for first-time learners. *Journal of Information Technology Education: Research*, 16, 209–226.

[36] Steimann, F. (2006). The paradoxical success of aspect-oriented programming. *ACM SIGPLAN Conference On Object-Oriented Programming Systems, Languages, And Applications (OOPSLA)*, 41(10), 481–497.

[37] Laddad, R. (2003). *AspectJ in action: Practical aspect-oriented programming*. Manning Publications Company. https://doi.org/10.1604/9781930110939

[38] Affandy, N. Suryana, S. Salam, M. S. Azmi and E. Noersasongko. (2011). 3De - Synergetic Program Visualization: A visual learning-aid tool for novice students. *International Conference on e-Education, Entertainment and e-Management*, 133–137, doi: 10.1109/ICeEEM.2011.6137862

[39] Diehl, S. (2007). *Software visualization: Visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media.

[40] Bentrad S. and Meslati D. (2011). Visual programming and program visualization-towards an ideal visual software engineering system. *ACEEE International Journal on Information Technology (IJIT)*, 1(3), 56–62.

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

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

[43] Eclipse AspectJ Project, https://www.eclipse.org/aspectj/

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

[45] AOSD community home page, http://aosd.net/

[46] Blockly, https://developers.google.com/blockly/

[47] Raptor, http://raptor.martincarlisle.com/

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

[49] Eclipse Modeling Project (EMP). http://www.eclipse.org/modeling/

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

Acknowledgment

The authors would like to thank the DGRSDT (General Directorate of Scientific Research and Technological Development) - MESRS (Ministry of Higher Education and Scientific Research), ALGERIA, for the financial support of LISCO Laboratory.

Authors Biography

Sassi BENTRAD obtained his Master degree in Computer Science and PhD in Software Engineering from the University of Badji Mokhtar-Annaba (UBMA), Algeria, in 2009 and 2015, respectively. Since 2015, he was appointed as an Assistant Professor at the department of Computer Science at the University of Chadli Bendjedid El-Tarf (UCBET). Currently, he is a researcher at the LISCO laboratory (Laboratoire d'Ingénierie des Systèmes COmplexes). His current research interests are Software Analysis and Visualization, Visual Programming, Model-Driven Engineering and Separation of Concerns.

Djamel MESLATI is a Professor in the department of Computer Science at the University of Badji Mokhtar-Annaba (UBMA), Algeria. He is the head of the LISCO laboratory (Laboratoire d'Ingéniérie des Systèmes COmplexes). His current research interests include Software Development, Evolution Methodologies, and Separation of Concerns.