Towards a New Way for Aspect-Oriented Software Development

Sassi Bentrad
Djamel Meslati
TOWARDS A NEW WAY FOR ASPECT-ORIENTED SOFTWARE PROGRAMMING

– VisPLAJ, a Pedagogic Visual Programming Language for AspectJ –

Sassi BENTRAD
Computer Science Department, LRI Laboratory
University Badji Mokhtar-Annaba
23000, BP 12, Annaba, Algeria
sassi_bentrad@hotmail.fr

Djamel MESLATI
Computer Science Department, LRI Laboratory
University Badji Mokhtar-Annaba
23000, BP 12, Annaba, Algeria
meslati_djamel@yahoo.com

Abstract:

Visual programming languages (VPLs) represent quite the biggest departure from traditional programming approaches. However, the last twenty years have seen quite remarkable progress in this field. While various visual programming tools have been proposed, there is a number already in the market and others are still in the research prototype stage, and it is difficult to predict the suitability of its usage for the real world applications. Their success is largely limited to specialised programming domains. Moreover, there has been less success in more general programming applications.

This paper offers a preview of our research project which we aim to evolve a new way for Aspect-Oriented Software Programming: Visual Aspect-Oriented Programming (VAOP) and to develop an educational language for teacher and student of this programming paradigm. Here, we present the current state of the art and emerging research in combining visual with aspect-oriented and object-oriented programming. It highlights some of the basic concepts of visual aspect and object-oriented programming, its classification, current research trends and the benefits gained from using it. This manuscript can be used to help researchers identify fruitful topics of future novice programming research.

Keywords: Visual Programming, Visual Language, Visual Aspect-Oriented Language, Aspect-Oriented Software Programming, Novice Programming.

1. Background and Significance

Programming is an abstract subject that is difficult to learn for most of novices. 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 [19], that they tend to approach programs line by line [23, 25], and that they are not able to handle the larger wholes of the programs [12]. Programming is known for its complexity and difficulty. It is believed to be hard to teach and to learn and many pupils and students in programming courses have difficulties to master all required competencies and skills. At introductory level of programming, that problem is even more notable.

The textual programming languages and environments have been developed ever since the computer was invented and therefore have been greatly influenced by the computer hardware organization. As a result, most of the textual languages developed are oriented towards using character-based input and output and the program structure is usually sequential. Beside hardware, programming languages are also influenced by natural language and mathematical formalisms such as algebra, predicate calculus and lambda calculus. This gives way to creation of variables to symbolize addresses, keywords such as if-then-else to describe conditional branches and unconditional branches. However, as computer progresses, it was found that some higher level languages do not favour natural language structure and choose mathematical formalisms like lambda calculus and predicate calculus to develop languages like LISP and PROLOG.

These languages which were developed as the result of the influences mentioned have several drawbacks. 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.

This requirement has restrained the programmer's thinking and forced him to consider organizing every program to be linear which may not be the requirements for its algorithm. 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. There are several other loopholes of textual programming such as their use of variables which have various conflicting roles, their weakness in expressing the concepts of object orientation and in describing data structures such as lists, graphs and trees which require many levels of abstraction from the intended semantics. However, text also has its worth which cannot be dismissed easily. Text is superior in expressing compact concepts such as algebraic formulae, comments and program element identification. As a result, it is always seen as part of the program.

Programming allows children or novice students to explore creative topics and learn problem-solving skills. Programming learning is complex for many novice students at university level. 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.

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 Aspect-Oriented Programming (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. These difficulties have prompted researchers to investigate tools and approaches that may ease the difficulty of teaching and learning programming. Popular novice programming tools include Scratch [14], Alice [5], and ToonTalk [9]. One of the most approaches that are being used for teaching programming is Visual Programming (VP).

Visual programming or tangible programming has considered as one of the promising research area to make programming easier for novices. Tangible programming systems reify programming constructs and objects as physical objects. It allows children to easily work together and move around as they program.

VP is a relatively new research and development area. Visual languages and visual environments aim to improve the programming process in different ways, but they both incorporate visual techniques as part of a strategy to achieve their goals. There is still much to learn about how to effectively use visual techniques to facilitate software development. So far, the most commercial successes have been in designing tools and visual techniques for constructing and navigating systems written in textual languages. This paper addresses the next steps towards successfully applying visual techniques to aspect and object-oriented software development.

Visual Programming Languages (VPLs) are becoming increasingly attractive and common in several domains. For example, VPLs are becoming the most common way to do some kinds of GUI programming, are becoming the most common way of specifying visualization graphics depicting scientific data, and are also starting to appear as macro generators for end-user applications. However, despite the increase in the use of VPLs for these and other programming tasks, there has been almost no attention to software engineering support mechanisms when working in these languages. VP is the use of graphical techniques in computer programming. It takes into account the need for programmers to communicate with computers using both graphics and text, and for computers to communicate with programmers.

In this section, we are looked at some concepts such as programming, traditional textual programming language, teaching programming and visual programming.

The rest of this paper is organized as follows: In section 2 we will present briefly our motivations. In sections 3, we will talk about aspect-oriented paradigm. Afterwards, in section 4 we will present the paradigm of visual programming. In the next section, we will discuss our current work. Finally, in section 6 we summarize the conclusion from our preliminary work and future work avenue.

2. Motivation

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 3D graphics toolkits like Inventor [24].
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 in computer science was noted and measured in the early 80's and again in this decade [22]. 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.

The need for new programming paradigms is a result of the ever-growing complexity of software. Object-Oriented Programming (OOP) is widely used in the software industry for managing large projects, but recently some of the weaknesses emerged. Problems like cross-cutting concerns, multi-dimensional separation of concerns are hard to handle.

Advanced separation of concerns brings multiple benefits for the software development. In effect, with new concepts, approach solves two difficult issues: the tangling and scattering of code. They are considered by software engineering community as the source of many problems and disposal therefore opens the way to improvements and promising outcomes for all the activities related to software engineering. Advanced separation of concerns 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 developers facing difficulties to design and implement programs including concerns.

Our idea is to appeal to the various techniques of visual programming to offer an approach that can contribute to simplify the work of the developer. It is question of a unified approach that can be used in combination with the main approaches for separations of concerns and in particular the aspect-oriented approaches. The work also includes the development of support tools that can simplify the software development.

Modern programming languages have made possible the birth of new paradigms like (C++) template meta-programming (TMP), generic programming (GP), and aspect-oriented programming (AOP) [10].

3. Aspect-Oriented Paradigm

Aspect-Oriented Programming (AOP) is one of the most promising new software development techniques. AOP is a relatively new programming paradigm based on the idea that computer systems are better programmed by separately specifying the various concerns of a system and some description of their relationships, and then relying on mechanisms in the underlying AOP environment to weave or compose them together into a coherent program. While the tendency in Object-Oriented Programming is to find commonality among classes and push it up in the inheritance tree, AOP attempts to realize scattered concerns as first-class elements, and eject them horizontally from the object structure. AOP aids a better handling of crosscutting concerns [11], as compared to object-orientation. Thus AOP aims to help in writing more modularized, and more maintainable code. Today's AOP implementations (among which the most widely-used is AspectJ), mostly rely on OOP. AspectJ essentially integrates tools for modularizing crosscutting concerns into object-oriented programs.

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.

AOP, a very complex and abstract approach to programming, has principles such as Join points, Point cuts, Advices and Aspects. Just reading about these concepts confuses the novice computer user, let alone programming using these concepts.

4. Visual Programming Paradigm

For a long time software developers have done their work using textual programming languages, but that is about to change. This section presents the paradigm of VP and VPLs as an alternative to improve the production of software applications.

4.1. Advent of Visual Programming

Visualization is long associated with computers and programming. Most of the analogue machines which were constructed prior to the development of digital computers were programmed in a pictorial fashion. Besides, flowcharting which uses picture like diagram, has been used as a heuristic aid in designing
algorithms in programming our present digital computers. Visual Programming field (or Graphical programming) has matured from the unification of progress in computer graphics, programming language and human-computer interaction. One of its pioneer contributions came from the work of Ivan Sutherland who designed Sketchpad on the TX-2 computer at MIT in 1963. His work has been recognized as the first computer graphics application which allowed users to work with a lightpen to create 2D graphics by creating simple primitives and applying operations on the geometric shapes which the users created. Sketchpad has contributed mostly in its graphical interface and support for user-specifiable constraints. The second major contribution in VPLs was made by David Canfield Smith in 1975 in his Ph.D. dissertation entitled "Pygmalion: A Creative Programming Environment". Smith introduced the idea of icon-based programming paradigm which allows the user to create, modify and link together small pictorial objects (icons) with defined properties to perform computations. Smith's work has initiated many other visual programming researches till today.

The main motivations for most research in VPL are: Many people think and remember things in terms of pictures. It is related to the world in a graphic inherently uses images as the primary component of creative thought. In addition, textual programming languages have proved somewhat difficult for many people to learn to use creative and intelligent effectively. The reduction or elimination of the need to translate visual ideas into text representations can help mitigate this problem the learning curve. In addition, a variety of applications, including scientific visualization and interactive simulation lends itself well to visual methods of development.

4.2. Visual Programming Languages (VPLs)

Visual programming (VP) based on the use of a visual language that manipulates visual information or supports visual interaction, or allows programming with visual expressions. VP refers to software development where the graphical notations and interactively manipulate software components are mainly used to define and compose programs. It is a set of spatial arrangements of text and graphics symbols with a semantic interpretation that is used to communicate actions in an environment.

A Visual Programming Language (VPL) can be defined as: A programming language that uses a visual representation (such as graphics, animations or icons, partially or completely). It uses visual techniques to express relationships or changes in the information [16]. VPLs often include text in a multidimensional way. Traditional textual programming languages sometimes also incorporate a little two-dimensional syntax for text, but only in very limited form. Thus, multidimensionality is the essential difference between VPLs and strictly textual languages.

Visual programming differs from textual programming in terms of expressing the syntax and semantics of programming languages. VP uses visual syntax which means that some terminals are able to display pictures or forms or other illustrations. A visual syntax may incorporate spatial information and visual attributes such as colour, depth and location. According to Shu [20], a VPL uses "some visual representations (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional programming language." Shu also emphasizes that "to be considered a VP, the language itself must employ some meaningful... visual expressions as a means of programming".

In our current scenarios, a VPL is usually embedded inside a visual environment. The environment is said to be a visual environment when the tools are graphical or adopted graphical techniques for manipulating pictorial elements and displaying program structure. This is where the programmer works to create, modify and design his programs. The environment may also consist of a set of tools and a user interface for accessing the tools.

The objective of having visual programming and visual environments is to enable programmers to express their logic or ideas of solving a problem in a simpler way, and to enable them to clearly understand how the program works visually. In order to achieve this objective, VPL usually uses the following four characteristics.

1. Conceptual Simplicity: VPL represents the underlying concepts as natural as possible and simplifies abstract concepts. It only emphasizes logic which is directly pertinent to the application and not the programming mechanics such as event loops, storage allocation and scope rule for objects.

2. Concreteness: VPL uses concreteness to facilitate creation of a program. Concreteness can be used to provide feedback or to provide specification to the program that needs to be created, for example, using

---

1 For example, in traditional languages, the x-dimension connects a linear string in the language, but the y-dimension may allow optional line spacing as a documentation device or for limited semantics (such as "continued from previous line"). Here only one of these dimensions truly conveys semantics, and the second dimension has been limited to a teletype notion of spatial relationships to be expressible in a one-dimensional string grammar.
3. **Explicitness**: VPL usually shows relationships among objects or modules in a very explicit way such as through connections or diagrams.

4. **Immediate Visual Feedback**: Liveness level refers to the degree in which VPL may provide immediate feedback. VPL is fully live as it displays the effects of changes made by the programmer automatically, that is, the programmer does not need to press a special button or do something in order to see the effects.

Given VPLs' unique features, the challenge is to develop software engineering approaches that are compatible with these features and take advantage of the opportunities they offer, yet have the power and rigor of traditional approaches.

### 4.3. Classification of VPLs

VPLs can be classified in different categories according to their general program representation paradigm [3].

As the field of VPLs has matured, more and more interest has been focused on creating a robust, standardized classification for work in the area. Such a classification system not only aids researchers in finding related work but also provides a baseline with which to compare and evaluate different systems. Some of the most important names in the field, including Chang, Shu, and Burnett, have worked on identifying the defining characteristics of the major categories of VPLs [4, 21, 3]. Note that these categories are by no means mutually exclusive. Indeed, many languages may belong to more than one category. They include, among others, we presents a summary of the each classification scheme below:

#### 1. Purely Visual Languages

The single most important category has to be purely visual languages that have been derived entirely or predominantly from graphical rules. Such languages are characterized by heavy reliance on visual techniques throughout the programming process. The programmer manipulates icons or other graphical representations to create a program which is subsequently debugged and executed in the same visual environment. The program created is compiled directly from its visual representation and is never translated into an interim text-based language. Examples of these languages include Pictorial Janus, VIPR, Prograph and PICT. There are other suggestion to subdivide this category into more specific language paradigm such as object-oriented, functional, imperative and logic.

Burnett and Baker [3] have developed a classification scheme for classifying VPL research papers. Their aim is to help other researchers to easily locate relevant works in this field.

#### 2. Hybrid – Text and Visual System

This subset of VPLs favours the idea of integration of visual languages with well-established textual programming languages. They perceive the integration might be more likely to meet the actual requirements of practical software development than the highly ambitious goal of creating purely visual languages. These hybrid systems may include both programs created visually and then translated into high-level textual language, or programs which involve the use of graphical elements in a textual language. The work of Andrew, Erwig and Meyer [6] are examples of programs created graphically and the system generates textual program from it. The latter includes works to extend existing textual language such as C++ and Basic. The current commercial system in this category includes Visual Basic and Visual C++.

#### 3. Others, Such as Constraint-Oriented Systems and Form-Based Systems

Beside the two major categories discussed, there exist many VPLs which fall into smaller classifications similar to Pygmalion or Sketchpad. They are termed constraint-based languages which are especially popular for simulation design. Constraint-oriented systems have also found application in the development of graphical user interfaces. An example of this is ThingLab II [13], which is an object-oriented and constrained programming system. A few other VPLs follow the metaphor of spreadsheets and these languages are classified as form-based VPLs.

Programming using form-based VPLs involves altering a group of interconnected cells and allowing the programmer to visualize the execution of a program as a sequence of different cell states which progress through time. Examples of form-based VPLs are Forms/3, ASP (Analytic Spreadsheet Package), and Penguims. Besides that, there are VPLs such as Vampire [15], ChemTrains and BITPICT which combine visual object-oriented programming language with a rule-based approach.

It is important to note that in each of the categories mentioned above, we can find examples of both general-purpose VPLs and languages designed for domain-specific applications. Table 1 summarizes classification of most of the visual system or VPLs frequently quoted according to the three categories highlighted above.
The field of visual programming has evolved greatly over the last twenty years. Continual development and refinement of languages in the categories discussed above have led to some work which was initially considered to be part of the field being reclassified as related to but not actually exemplifying visual programming. These VPL orphans, so to speak, include algorithm animation systems, such as BALSA [2], which provide interactive graphical displays of executing programs and graphical user interface development tools, like those provided with many modern compilers including Microsoft Visual C++. Both types of systems certainly include highly visual components, but they are more graphics applications and template generators than actual programming languages.

### 4.4. Benefits of Visual Programming

Visual programming addresses two important issues in software engineering. Firstly, it enables users to master the complexity of programming by visualizing it. The raising of the abstraction to the visual level reveals semantic relationships among program entities and makes it more understandable. Another one is increasing productivity which is gained either through the ease of using the systems or the increase in communication among the developers and the users of the systems, because the users can usually understand the beginning process of designing the systems.

Some studies have been carried out to compare visual programming with other types of programming. Green [7] compared the readability of textual and graphical programming and concluded that graphical programs took longer to understand than textual ones. Moher [17] also compared petri-net representations with textual program representation and found areas where the petri-net representation was more suitable. Besides, Pandey and Burnett [18] compared time, ease, and errors in constructing code using visual and textual languages. They found that matrix and vector manipulation programs constructed using visual programming had fewer errors. Another study by Cunniff and Taylor on the comprehension of a static visual flowchart language versus the textual language Pascal was conducted on novice programmers. The finding is based on reaction time and on the number of correct responses from the programmers. The study reported that the flowchart language was easier to comprehend.

In the real world application, Baroth and Hartsough [1] cited several advantages of visual programming such as its flexibility in the design process, improvement of communication between users and programmers, and shorter period to train programmer to master the language. Besides, they have observed increased productivity through a reduction in software development time as communication between the customer, the developer and the computer is facilitated by the visual programming tools used.

Other attractive features of visual programming tools are that they are easy to use and understand and make rapid prototyping possible. As a result, early testing and demonstration, and allows corrections to be made early in the development life cycle. The prototype can later be refined to produce the final application. Visual programming also makes coding easier to follow and the codes become more reusable, easier to debug and easier to document. Maintenance is also easier because the visual programs are usually runtime reconfigurable. As a result, the pace of work is faster since changes made visually are immediately in effect and testable without the recompilation stage. On the whole, it shortens up the development process and also enables the users to be involved more directly in the process of programming. Future research should focus more on its use in the development of applications.

### 4.5. Drawbacks of Visual Programming

The problems mentioned in this section apply to many visual programming.

- Difficulty with large programs and large data, too much information to fit on the screen
- Need for automatic layout
- Unstructured programs
- Static representation of programs that are hard to understand
- No place for comments
- Lack of formal specification
- Lack of functionality
- Lack of efficiency and evidence of their worth
- Tremendous difficulty in building editors and environments
- Poor representations
• Lack of Portability of Programs

5. VisPŁAJ, a Pedagogic VPL for AspectJ

5.1. Discussion

Aspect-Oriented Programming (AOP) is an evolving paradigm for software development programming. It is based on the realization that different parts of a program are best implemented using a different set of techniques; this is called separation of concerns [11]. AOP, a relatively new programming paradigm, recently earned the scientific community's attention. Having around six years of life, the paradigm was already presented in important conferences, and recently triggered the creation of several conferences and workshops to deal with it.

Nowadays AOP is widely used in both academic and industrial world. Practice shows that AOP programs are in many cases shorter, have more modular structure and are easier to understand. Numerous publications discuss the advantages of AOP design and implementation. The complexity of an AOP program depends on the OOP components and the AOP-specific constructs. Therefore the complexity could be scattered between the AOP-specific parts (in pointcut-definitions, advices, etc.), the object-oriented constructs (classes, inheritance, etc.), and even in the procedural-style implementation of the methods.

Designing programming environments for novices is a fascinating field that we have only just started to explore. There are a great many more questions and answers to explore, and some wonderful environments yet to build and try in that exploration. The progress in the field is toward making programming more interesting, more relevant, and more powerful for students. The research opportunities could hardly be broader, and promise enormous potential impact.

The research has considered what advantages were to be gained from using visual programming in this educational context and has sought to identify the source of those advantages.
• Visual languages by their design can provide a simple syntax that makes them easier to use than textual languages with a more complex syntax.
• While the visual programming language being investigated appeared to facilitate programming for those with little experience, it is nevertheless important that support materials and teaching dialogue uses consistent and clear descriptions throughout.

Instead of thinking of VPLs as a future replacement of general purpose programming languages, we may have to let it evolve and see if we can find useful niches for it. A VPL is totally different from any linear, symbolic language and perhaps will visual languages in the future allow us to do things that were not possible with today's languages or perhaps do it in a way that will be more obvious and intuitive for non-programmers.

Although the VPLs communicate information in a more visual than traditional textual languages, the text still has its place in the VP. There are three areas that illustrate the superiority of the text in some situations: for documentation, assign names to distinguish between elements of the same type, and express well-known concepts are inherently compact and textual (e.g. algebraic formulas).

The idea of our research project is to use various techniques of visual programming to provide an approach that can help to simplify the developer's work.

There is talk of a unified approach that can be used in combination with the main approaches of separation of concerns and especially aspect-oriented approaches. The future work also includes the development of support tools that can simplify the software development.

Visual Aspect-Oriented Programming (VAOP) is an extension of AOP in which some aspect languages are visual. It provides the opportunity to improve development of component software through better interfaces, yet still producing code with the same level of efficiency as C by raising the level of abstraction. However, abstraction often results in less efficient code. To offset this tendency, domain specific abstractions must be used. While some generality is lost, domain specific abstractions are more efficient to implement.

VAOP is an emerging area that combines both aspect-oriented and visual programming techniques. Common usage applies this term to two distinct contexts, either to the use of an aspect-oriented programming language that has a visual syntax, or to the use of an environment with graphical tools to manipulate programs written in a textual aspect-oriented language.

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 programming environment for teaching AOP is, therefore, desirable.

To this purpose, the aim of the first phase in our research project is to develop a pedagogical visual programming language to support learning of basic concepts of AOP with AspectJ.

5.2. Current Work

Our work focuses on the Hybrid (or Heterogeneous) Visual Programming approach (HVP) to develop a VAOP environment specifically for AspectJ. We call it VisPLAJ. The name VisPLAJ is an acronym for "Visual Programming Language for AspectJ", the language together with the environment are an attempt to integrate the visual and textual programming paradigms for one of the most Aspect-Oriented Programming (AOP) approaches applied to the Java language— AspectJ [6]. The prototype environment presented here is currently being implemented as a set of plug-ins under Eclipse platform.

HVP integrates visual and textual expression in programming. Therefore, the concept of Hybrid Visual Programming Language (HVPL) allows integrating a visual sub-language V into a textual language L such that V-expressions can be used as substitutes for certain L-constructs. The HVPL environment translates the V-expressions used in a program into their textual equivalents and reintegrates these translations with the textual parts of the original source program according to an extended grammar for L. As the target language for picture translation is as well L, the compilation output for a heterogeneous program written in L+V is a textual program L-program. This, of course, can be processed by any standard interpreter or compiler for L.

A side advantage of this scheme is that we can even modify and revise existing programs written in L using visual expressions from V. The resulting heterogeneous programs can then be reprocessed and can again be translated by the original compiler for L. Thus, we achieve a way of maintaining and extending programs in a heterogeneous visual language which have originally been written in a textual language. Hence, there is no need to discontinue the maintenance of existing software when migrating to a visual environment.

A. Design

In our approach processing, the execution the VisPLAJ program is achieved in three phases.

First, the heterogeneous program (i.e. V+L program written in L+V) is split into pictures and textual program and then replacing all the pictures by unique identifiers. Additionally, a database of pictures indexed by these identifiers is created. The second phase is the translation of each visual term (extracted from pictures) into its textual equivalent. In the last phase, the converted textual source code is parsed according to an attribute grammar for L+pictures and is then recombined with the picture’s textual equivalents by means of syntax directed translation (an ordinary textual parser). The resulting L-program (i.e. a standard AspectJ program) is processed by a standard compiler for L (i.e. AspectJ language).

B. Implementation

Implementing a visual programming language (VPL) is much harder than implementing a textual language. VPLs are usually embedded and tightly integrated within visual environments. Consequently, they are often characterized by the attributes of the environments [1]. The VPL implementation involves the implementation of a whole programming environment with a user interface which supports developing programs using a visual language. Notice that VPL interfaces are not the same as graphical user interfaces (GUIs) nor are they just for visualization. Traditional GUI development toolkits are inadequate for the creation of VPLs because they do not support syntactic and semantic specifications of visual programming. The graphical user interface of a visual language relates to the language's syntax and semantics. The interaction (dialogue) between the interface, the syntax, and the semantics must be maintained. Implementing a VPL interface and its support for syntactic and semantic specifications of visual programming suffers from a problem common to all large, complex software systems.
Besides technical issues, one of the main concerns when defining a VPL is finding a balance between the \textit{expressiveness} and \textit{implementability} of the language. 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 taking this approach (HVP) to mix the two in software development, a textual programming language is used to describe the components, and visual language is for expressing the structure and relations, i.e. the architecture.

In our research 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).

6. Conclusion and Future Work

A new way for aspect-oriented software programming and a preliminary work to develop a pedagogical visual programming language for AspectJ are presented and discussed in this research paper.

The aim of designing this language is to make programming simple, easy and more understandable to the audience who are fresh or new in programming, to improve the correctness with which people perform programming tasks, and/or to improve the speed with which people perform programming tasks. It provides ease to programmers and more intuitive for novices, i.e. they are not required to learn or remember syntax. They will just have to enter the parameters using visual constructs. It is an attempt to make programming and maintenance of programs easier, and therefore cheaper.

A hybrid technique is used to design VisPLAJ, both visual and textual expression will be used to construct a program using this language. Throughout its design we are considering the aspect of understandability, easy to program, readability, writability and correctness of the language.

VisPLAJ will hopefully make programming less frustrating, more productive and also we hope to be used for AspectJ language learning for beginners and to be helpful for the rapid development of small AspectJ programs.

This new approach is an excellent way to teach programming and solve problems, and suggested 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 language, VisPLAJ, will open a new-direction in the aspect-oriented software programming area.

References:


