Towards a Hybrid Approach to Build Aspect-Oriented Programs

Sassi BENTRAD, Hasan KAHTAN KHALAF, and Djamel MESLATI

Abstract—**The success of new programming paradigm such as the Aspect-Oriented Programming (AOP) relies mainly on solid support tools and advanced development environments. However, the productivity is still restricted by a text-based primary input method at coding level, what makes program understanding, building, and maintaining difficult for developers. To reduce such difficulties, we propose to take the AOP out of the conventional style of coding by using a new approach, which is partially visual. The approach is intended to better support the coding process by introducing more interactivity and a high-degree of flexibility. We seek to minimize the influence of language syntax on overall usability by using the ordinary drag-and-drop technique to overcome the weaknesses of text-based style to the of AOP paradigm. Our approach has been implemented in an Eclipse-based prototype tool and evaluated through a controlled experiment to prove its feasibility and usefulness. As preliminary results, we notice that programmers were able to express effectively crosscutting concerns with a high-level of interactivity.**

Index Terms— **Aspect-oriented programming (AOP), AspectJ, Coding methodology, Codeless program development, Separation of concerns (SoC), Visual programming.**

I. Introduction

In the software development life cycle, the implementation often referred to "programming" or "coding", is regarded as one phase, where generally, the adopted method depends on writing source-code by hand according to a specific language syntax. For a long time, programmers have done their work using tools that depend on the text-based style, and were often confronted with the difficulty of evolving their codes. During the understanding process, they may execute several tasks all together such as *reading*, *searching*, *thinking*, *translating*, *recall* and *mental modeling*, which make much harder the focus on specific problems.

Programming languages are the primary vehicles for supporting the practices of software engineering. To address various issues, it is important, therefore, that they should be well designed and implemented along with their supporting tools [35]. The latter must offer simultaneously a high-degree of flexibility and efficiency in code editors, so that making the programming process more efficient. There are many different attempts and a major effort has been directed to overcome this challenge. The modern editors come with some helpful features like *code outline*, *syntax coloring*, *highlighting* and *checking*, *code auto-completion*, and so on, in order to make the traditional programming style less disheartening and boring, especially for novices having only basic understanding of concepts and programming-language constructs. However, usually most of abstractions that are meaningful at the design may be lost when implemented at the coding phase.

In spite of these advances, using text-based editors still requires programmers to spend effort and focus on implementation details. A considerable research issues have been identified for making the act of "coding" relatively easy and effective, and researchers are focusing on bringing more improvements to the coding process [45]. 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) [4,9]. In the last few years, the tendency of programming environments to support graphical techniques has been emphasized to provide the development, the execution and the visualization of the programs [8,9]. Unfortunately, most of them have proven success within limited domains as the case of *Visual Zero* [14], *Tersus* [27], etc. Most recently, a significant attention from the research community has been given for assisting the general-purpose programming tasks. The codeless program development represents one direction for prototyping and building quickly programs in a high-level of interactivity. It is a convenient way that tries to lessen the focus on formalisms by exploring the idea of code structure editor [26].

Aspect-oriented programming (AOP) is a dynamic research field that focuses on the modular implementation of concerns (i.e., *non-business operations* such as: logging, authentication, threading, transactions...) that cut across a system's functionalities (i.e., *business logic*) [2,10,17]. It came to provide and to deliver a better separation of concerns (SoC). *Gregor Kiczales* coined the term as a complement to the object-oriented programming (OOP) paradigm rather than as a replacement to it [2]. However, there are still some problems, like implementation cost and its complexity were not in view, that defined by *Gail C. Murphy* as the main factors to keep in mind while evaluating

Manuscript received November 1, 2019; revised July 30, 2020. This work was supported in part by the DGRSDT (General Directorate of Scientific Research and Technological Development) - MESRS (Ministry of Higher Education and Scientific Research), Algeria.

Sassi BENTRAD is with the LISCO Laboratory, University of Badji Mokhtar-Annaba (UBMA), and the Department of Computer Science, University of Chadli Bendjedid (UCBET), Algeria (e-mail: sassi_bentrad@hotmail.fr, bentrad-sassi@univ-eltarf.dz).

Hasan KAHTAN KHALAF is with the Faculty of Computer Systems and Software Engineering, Universiti Malaysia Pahang (UMP), Malaysia (e-mail: hasankahtan@ump.edu.my).

Djamel MESLATI is with the LISCO Laboratory (Laboratoire d'Ingénierie des Systèmes COmplexes), University of Badji Mokhtar-Annaba (UBMA), Algeria (e-mail: meslati_djamel@yahoo.com).

a software engineering methodology [50].

As with any new technology, AOP has both strengths and limitations in terms of their impact on software engineering. *Roger T. Alexander* and *James M. Bieman* [21] reported a number of studies that explore these challenges. *Muhammad Sarmad Ali et al*. [37] have performed a systematic literature review of empirical studies that explore the benefits and limitations of AOP-based development from the perspective of its effect on certain characteristics. According to their findings, a majority of the studies reported positive effects for code size, performance, modularity, and evolution related characteristics, and a few studies reported negative effects, where AOP appears to have performed poorly on cognitive dimensions of software development (i.e., cognitive burden issues) due to the new language constructs and mechanisms offered. Cognitive outcomes were measured by looking at two relevant factors: the time taken for understandability and development efficiency, which is measured in terms of the amount of time and effort spent to build programs. Obtained results were insignificant and not encouraging [18].

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 [36]. The reasons that have hindered its wide acceptability are: (1) the awareness (it is still less user friendly); (2) the lack of universal supporting framework; and (3) it has been still less heard of so technical experts are very few in number [21]. In addition, AOP introduced new dimensions and standards to programming. This, in general, creates complexity and possible resistance, but it was also the case when OOP was introduced, which indicates that this is a normal scenario [15,21].

Over the last few years, it has matured and received increasing attention from researchers across the world. Numerous works has been carried out on strong AOP-based implementations such as AspectJ language [3,20]. However, their acceptance in mainstream software development is still limited [36]. They are mainly used only for maintaining, rather than for developing the initial version of a system [19]. The prominent reason for this is the fact that support tools purely depend on the text-based style, which generally do not facilitate development tasks, as is the case of AJDT (AspectJ Development Tools [10,24]) in spite of its completeness and maturity.

The success of such new paradigm heavily relies on providing a solid support that allows its adoption in industry and provide the basic resources for both developers and researchers to study and understand it. We do agree that in training, understanding the abstract model of a programming language is more crucial than its syntax. However, novices spend usually considerable time and efforts for learning formalisms and following strict rules to ensure that the source program executes [39]. This, decreases relatively the motivation to programming and understanding the essence of the paradigm adopted, and even precludes them from being more creative [36].

With new powerful language constructs that are not straightforward, AOP offers new ways of implementing traditional mechanisms. For novices, expressing and specifying crosscutting concerns and their relationships is considered to be a challenging issue that existing support tools, such as AJDT, cannot assist to express it very effectively. Certainly, this area needs further research. In this article, we introduce a new approach to handle this issue. We target to provide better results in code quality and software development efficiency.

In our opinion, even if it is not ready for large-scale industrial adoption, since most supporting tools are still being in infancy stage of development [44], the visual programming (VP) capabilities may help in maturing AOP and makes it worthwhile to receive more attention in both academia and industry [40,42,47]. Therefore, to better serve the needs of improving the quality and usability of tools, we should devise new form for programming that allows code and visual objects to be freely combined with more interactivity and flexibility. The main aim is to providing a higher freedom degree with respect to source-code languages and their syntax difficulties. Therefore, programming efforts go into interacting with graphical interface instead of focusing on typing codes. This reduces the amount of code that has to be written, in addition to assisting in understanding and handling efficiently the source-code, what consequently, reduces the time-consuming and effort.

In this work, we present a new approach for building aspect-oriented (AO) programs in which both text-based and drag-and-drop methods are used conjointly for taking more advantages of their capabilities. The goal of our proposal is to investigate the effect of a hybrid interactive technique on AOP tasks, and how the graphical expression of basic programming constructs and features provides a new opportunity for programming in an efficient way with simplicity and high-degree of flexibility. We seek to raise awareness of the strengths of AOP technology, and to decrease knowledge required for its use.

Our research work aims to achieving two goals:

- We aim to provide a new way of AO-specific programming facility, allowing programmers on one hand, to express crosscutting concerns using visual representations of concepts (*visual code development*), and on the other hand, making the program source entities and their relationships, explicitly visible and accessible by constructing visualization views (*graphical code illustration*). This allows assisting them in understanding and handling the overall source-code efficiently during development and maintenance activities. This work is the beginning of an ongoing project to introduce a new methodology for improving AOP coding skills and increasing its adoption.

- Our initial work focus is on the educational context where we provide a preliminary supporting tool addressing the AspectJ implementation for training purposes. The prototype is expected to assist novices in building knowledge on AOP concepts and features and being familiar with them. The main goal is to allow them building effectively programs with more focus on innovations rather than on implementation details such as syntactic formalisms. This will allow obtaining high-quality codes with less miss-typing, which further will simplify both teaching and training in terms of amount of time and effort spent for understanding and programming.

The remainder of this article is organized as follows: Section II presents an overview of relevant background to VP capabilities and AOP. Section III is devoted to introducing the proposed approach, the process and technologies used to develop it, followed by a detailed description of the design and implementation of the tool support. Section IV reports and discusses the results of a case study on an illustrative example and a preliminary user evaluation. Section V describes an overview of some related works. Finally, in Section VI, conclusions are drawn and further work issues that we plan to investigate are given.

II. Background

According to *Nong Ye* and *Gavriel Salvendy* [35], technical experts have better knowledge of programming at an abstract level, and novices tend to have more concrete knowledge. Current research works seek to provide a higher-level of abstraction for developers by exploiting various graphical techniques during the development process [8]. They tend to make programming tasks easier for those having little background in the field, and may also be useful for the experienced ones for a fast software development or prototyping.

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 [8,9,40]. The main reason for using such techniques is that they are often more convenient to users than the traditional text-based style. It allows representing the coding itself entirely or partially using graphical constructs instead of, or in addition to, the text-based coding [9]. In many cases, handling interactively visual representations offer significant advantages (for comprehension and development of large systems) over textual descriptions [45].

However, there is a common misunderstanding, which assumes that the research targets to eliminate the text-based method. In fact, this is a fallacy; most visual programming languages (VPLs) include text to some extent, in a multidimensional context. Their overall goal is to strive for improvements in the design of programming languages and associated tools. The opportunity to achieve this comes from the fact that in VP, we have fewer syntactic restrictions on the way a program can be expressed interactively, and this affords an independence to discover programming mechanisms that have not been possible formerly [8,9,42,46].

On the other hand, unlike the text-based style that can be used for any coding tasks, the visual style is only suitable for certain tasks (limitation of suitability). For instance, in some cases of complex control structures like loops and recursion, the textual description is often more efficient and economic, and the code is usually more compact than visual programs.

SoC is an important software engineering principle, meaning the ability to identify, encapsulate, and manipulate those parts of the software that are relevant to a particular concern (concept, goal, purpose).

Although it is an abstract concept, a concern at implementation level is usually considered as a particular behavior or functionality in a program. Concerns can be very primitive, such as adding a variable. High-level concerns are coarser, such as transaction management.

A new emerging paradigm is introduced–AOP that makes possible to build those programs that OOP fails to support. It deals with those concerns that cut across the modularity of traditional programming mechanisms [17]. AOP languages have been an important means to control the complexity, to improve the modularity and to support development flexibility. A good overview of the AOP scene can be found at [22].

It has emerged initially at the programming level using strong implementations such as AspectJ, the de facto AOP standard language [2,10]. AspectJ encapsulates crosscutting concerns into new modular programming abstractions called "*aspects*" to preserve modularity instead of scattering them in the core modules "*classes*" [1,2]. However, programmers and especially novices experience some difficulties in using syntactic formalisms of some concepts and features [36].

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

III. Our Approach

A. An Overview

For the SoC we distinguished two different levels. The concerns identified at the conceptual level, generally considered a primary means to manage complexity, are mapped into the implementation level using a programming language. By abstracting concerns out and separating them, implementing individual concerns becomes substantially less complex, and code can be effectively reused. However, few languages allow these meaningful abstractions to be separately implemented [52].

Especially for novices without highly technical backgrounds, AOP complicated programming by combining two programming levels; for the low base code and crosscutting concerns. Our opinion on this issue is the use of a hybrid approach, a visual and text-based oriented method. It is a seamless integration between both to support novices' difficulties at coding time when combining these two levels.

The idea of codeless program development, such as under the tool *Limnor Studio* [26] whither it enhances the OOP by adding actions for reducing the hand-typing, is to make a general-purpose visual style that is simple and preserves the programming power. The trend was towards moving farther away from traditional editors through elevating the level of abstraction. The associated tools are changing the role of software engineering and allowing novice programmers to more easily developing and even getting quick overviews of large source-codes, which is difficult without higher abstractions. It is likely that this kind of tools will make the next-generation of development systems. Significant advantages can be mentioned as follows:

i. Developers can focus more on the design and innovations,

ii. Higher quality of code by avoiding potential and the most common programming mistakes, and

iii. Less time and effort to accomplish tasks efficiently, this boosts the productivity for huge systems.

This is the source of our inspiration to propose a new kind of code editing. Our proposal addresses particularly the aspect-oriented approach. The idea we present in this article is a **H**ybrid **M**ethodology **for A**spect-**O**riented **P**rogramming (**HM4AOP**). It provides a way to leverage the usage of interactivity during the coding process, which can help bridge the gap among programmers and the complex syntactic formalisms on programming system, while overcoming some weaknesses of the text-based method.

Before editing the target source program in an AOP-based implementation (e.g., AspectJ), and as an essential step, the skeletons of code have to be designed and specified interactively in a high-level way by using predefined visual representations and specifications for the fundamental AO-basic programming constructs and features. We retain the strengths of text-based style, while enabling the visual way where it is beneficial. While graphical techniques are cumbersome to work with in general-purpose programming; we do agree, that it is probably best suited for a specific-domain, whereas text-based languages may be used as host-languages, and carefully designed graphical notations can be useful.

In our proposal, we have to provide programmers with a high-level way to express the domain-specific concepts and features of interest, and isolate the low-level implementation concerns, so that even non-professional programmers can prototype and efficiently create programs. More specifically, they can describe the structure and entities of base code (*business logic*) besides crosscutting concerns (*non-business operations*) and their relationships in a flexible combination of textual and graphical notations within an appropriate editor.

Figure 1 illustrates our proposal. It depicts the overall programming process, in the case of the conventional method adopted for coding and our methodology. We can distinguish two programming styles. The first one is a low-level presented via a host-language, a conventional text-based language. The second is a high-level presented via a domain-specific visual language with added concepts and features borrowed from the AOP paradigm.

The overall process of AOP include:

i. Developing primary abstractions (*base code*).

ii. Identifying concerns that crosscut primary abstractions.

iii. Defining aspects to encapsulate each concern.

iv. Weaving aspects into the primary abstractions, yielding a composite program compiled.

v. Executing the composite program (*woven program*).

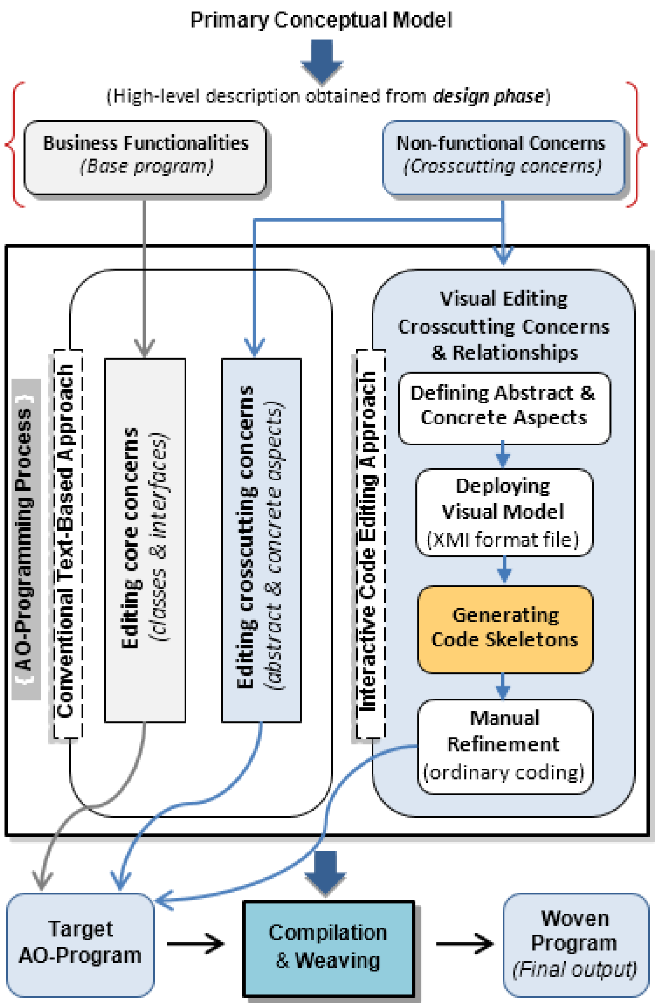

Primary Conceptual Model
(High-level description obtained from *design phase*)
Business Functionalities (*Base program*)
Non-functional Concerns (*Crosscutting concerns*)
{ AO-Programming Process }
Conventional Text-Based Approach
Editing core concerns (*classes & interfaces*)
Editing crosscutting concerns (*abstract & concrete aspects*)
Interactive Code Editing Approach
Visual Editing Crosscutting Concerns & Relationships
Defining Abstract & Concrete Aspects
Deploying Visual Model (XMI format file)
Generating Code Skeletons
Manual Refinement (ordinary coding)
Target AO-Program
Compilation & Weaving
Woven Program (*Final output*)

Fig. 1. Overall process of a hybrid construction of aspect-oriented programs.

By doing so, programmers will obtain the benefits of both the widely used general-purpose language (i.e., *code-oriented method*) and the straightforward domain-specific language (i.e., *interactive code editing*). An advanced visual editor will be designed to make them more productive editing code; specifically for crosscutting concerns and their relationships. We try to reduce cognitive load while coding by simplifying as much as possible. It is important, to notice that the programmer will be able to switch to whatever coding style is appropriate to a given context. Therefore, both textual and graphical techniques are complementing each other, and we can take more advantages of each one and avoid their limitations.

B. *The Approach*

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

A brief comparison between the conventional code editing and our proposal is shown in Table 1.

TABLE 1
CONVENTIONAL CODING VS. HYBRID CODELESS METHODOLOGY

Conventional Input Methodology (Text-based programming)	**Hybrid Codeless Methodology** (General-purpose partially-visual code editing)
Ordinary Code Editor (OCE)	Visual Model Designer (VMD), an advanced visual editor
Code-oriented method (i.e., hand-typing)	Interactive code editing method, a flexible combination of the text-based style, graphical constructs and simple objects.
Syntactic formalisms of programming language	Visual representations & specifications to build a *Visual Model* of target program (i.e., a high-level description and specification of source-code structure).
Writing code entirely within a text-based editor	Interaction techniques (e.g., drag-and-drop interaction) on top of a general-purpose hybrid programming interface.
Ordinary Compiler such as the *AspectJ Compiler* (ajc)	Visual programming components (modules): (1) **Model constructor (VMD)** to build a *Visual Model* of target program ; (2) **AO code-generation engine** to insure the translation service of the *Visual Model* into *Code Skeletons Templates* (i.e., a source-code structure in text-based form).

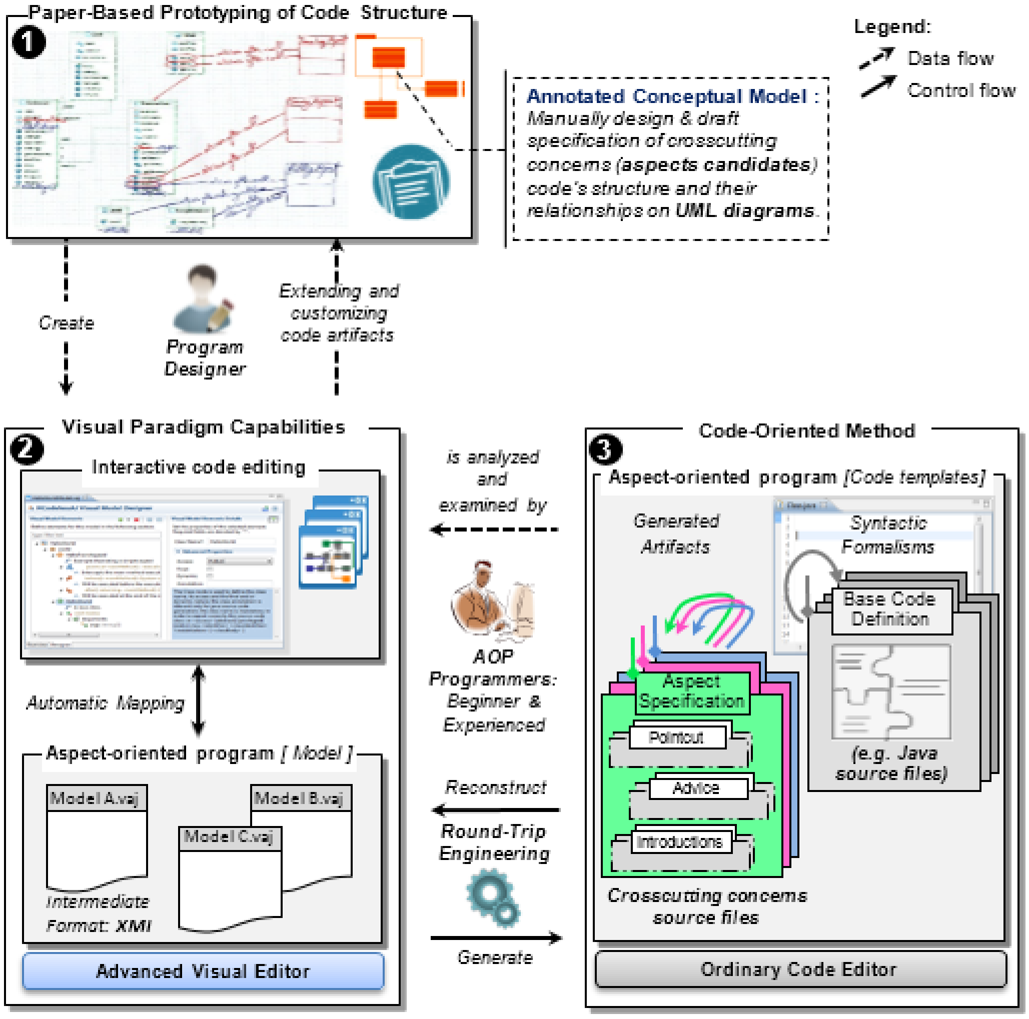

Paper-Based Prototyping of Code Structure
1
Legend:
Data flow
Control flow
Annotated Conceptual Model :
Manually design & draft specification of crosscutting concerns (aspects candidates) code's structure and their relationships on UML diagrams.
Create
Program Designer
Extending and customizing code artifacts
2
Visual Paradigm Capabilities
Interactive code editing
Automatic Mapping
Aspect-oriented program [Model]
Model A.vaj
Model B.vaj
Model C.vaj
Intermediate Format: XMI
Advanced Visual Editor
is analyzed and examined by
AOP Programmers: Beginner & Experienced
Reconstruct
Round-Trip Engineering
Generate
3
Code-Oriented Method
Aspect-oriented program [Code templates]
Generated Artifacts
Syntactic Formalisms
Base Code Definition
(e.g. Java source files)
Aspect Specification
Pointcut
Advice
Introductions
Crosscutting concerns source files
Ordinary Code Editor

Fig. 2. Global view of the proposed approach.
(Numbers indicate the sequence of process steps in building & rebuilding an aspect-oriented program)

Figure 2 illustrates the idea we propose and its practical supporting tool. We aim to introduce a next-generation of tools for making development process more attractive and flexible. It features an effective and scalable interface that improves the ability to view, edit, and interact with code visually. The process is focused on the building of the program structure, behavior and requirements. As shown earlier, the building process can be summarized in the following three steps:

The first step (labeled ❶ in Figure 2), the "*Rapid Code-Prototyping*", requires one to design manually preliminary templates of AO code skeletons for the program being developed and specify the structure of its entities and their relationships. The composition specification depends on some expressive artifacts using the conventional natural language as a simple scripting language on sheets of UML diagrams, without any structural definitions or syntactic formalisms. This paper-based prototyping can be considered as an initial implementation that helps to find design faults especially for the non-functional concerns (*aspects candidates*) and their relationships at an early stage (i.e., specification and design phases).

In the second step (labeled ❷ in Figure 2), the "*Interactive Code Editing*", the programmer expresses the target code structure and some behavior according to these draft templates in terms of visual, interactive elements within a visual editor. Once the visual coding is accomplished, the resulting model can be inspected, controlled manually and discussed through high-level views.

These two steps constitute the quickest way to design the preliminary templates, as well as a key method to empower programmers to:

i. Maintain the separation of concerns earlier in the life cycle.

ii. Refine and make meaningful changes quickly in the code,

iii. Communicate seamlessly —involve development team members to get feedback early in a collaborative environment,

iv. Be more creative —experiment with many ideas before committing to one, and

v. Save programming time by solving key problems from the very beginning —the unexpected changes can be extremely expensive to be implemented when they are discovered later at coding time, in contrast, the early identification of what is really required to achieve high-quality codes can lead to more productivity, especially for large software applications.

To perform the code generation and to simplify interchanging of the constructed models among developers and their favorites IDEs with manipulation capabilities, such models must be available in a standard and interoperable format, a text-based notation that can be readily processed. This is possible using the XMI (*XML Metadata Interchange*) standard. The main purpose is to serialize the visual model data in an XMI format file (considered as an intermediate model), which can then be easily deployed and manipulated automatically, without the need of intermediate models, as usually done in semi-formal approaches. This file can then be fed to the suitable generator for producing on output AO code templates.

At the third step (labeled ❸ in Figure 2), the "*Ordinary Coding*", the generated templates can then be manually refined and re-edited by adding the required code (i.e., hand-made modification for completing both structural and behavioral codes of generated methods, advices, etc.) until obtaining an ultimate implementation for the target program.

Following the process outlined, we now describe, in Figure 3, the overall flow of main activities using an illustrative example of AspectJ program.

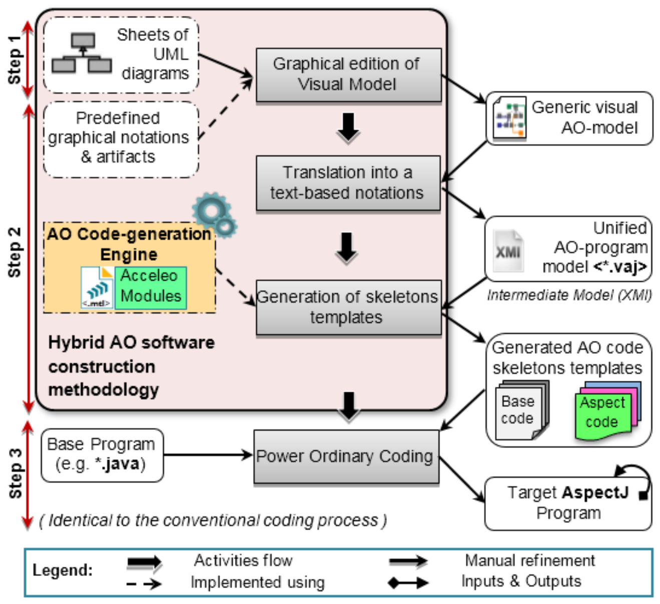

Step 1
Step 2
Step 3
Sheets of UML diagrams
Predefined graphical notations & artifacts
Graphical edition of Visual Model
Translation into a text-based notations
Generation of skeletons templates
AO Code-generation Engine
Acceleo Modules
Hybrid AO software construction methodology
Generic visual AO-model
Unified AO-program model <*.vaj>
Intermediate Model (XMI)
Generated AO code skeletons templates
Base code
Aspect code
Base Program (e.g. *.java)
Power Ordinary Coding
Target AspectJ Program
(Identical to the conventional coding process)
Legend:
Activities flow
Implemented using
Manual refinement
Inputs & Outputs

Fig. 3. Steps of coding process.

C. The Tool Support

We show the practical feasibility and effectiveness of the approach through a tool prototype over Eclipse IDE. We have chosen AspectJ to be our first target implementation as it is the most frequently used and mature. AJDT is arguably the most complete, open and mature support, additionally regarded as representative in the AOP research community [11].

We have so far developed a first prototype called "**HCodelessAJ**" (**Hy**brid **Codeless** Methodology for **AspectJ**). More precisely at technical level, it is a general-purpose icon-based coding tool, completely leverages AJDT and includes initial implementation. It offers a hybrid-programming interface with the following features: (1) users can present the structure of the program and some behavior in a flexible combination of textual and graphical notations within a visual editor; (2) visual representations are coupled with interaction techniques to simplify the navigation and understanding of code. Users can easily verify the completeness of entities that make up the program source and the consistency of its relationships; and (3) regardless of which development stage they are currently at, they can rapidly check what they have developed. This is crucial in modern tools, where we have to deal with huge systems and are subject to information overload from various programming tasks: *analysis*, *verification*, *testing*, *debugging*, and so on.

1) Design

Figure 4 depicts the overall structure of an open architecture for HM4AOP supporting tools, with the adopted implementation frameworks. Open means that there are no limits for both internal and external extensibility.

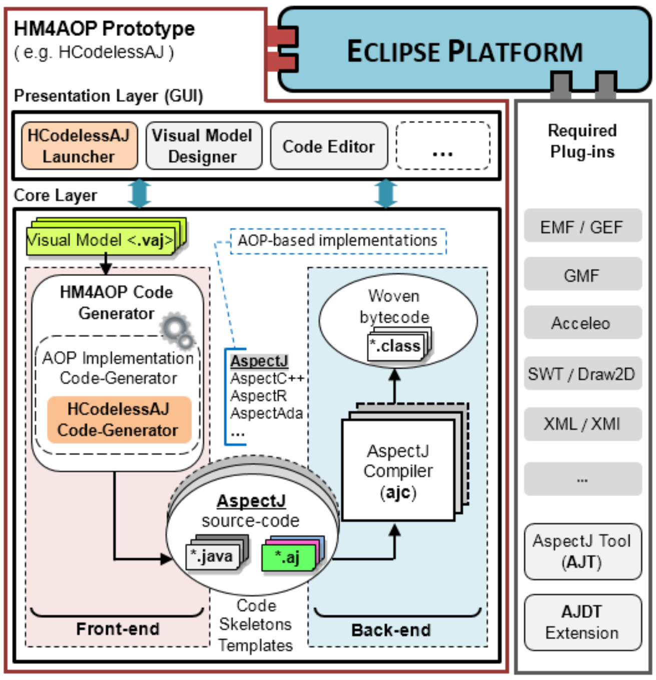


Fig. 4. High-level architecture of HM4AOP tool support.

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

- **HM4AOP Code Generator**: A specific code-generation engine, i.e., AOP implementation generator, represents this component. In our implementation, the *HCodelessAJ Code-Generator* is a *template-based AspectJ code extractor* based on Acceleo technology [30]. It takes as input the model already built, and produces as output textual templates of code suitable for the target implementation. In the following sub-section, we will explain the transformation process in more detail.

- **HM4AOP Model**: This *Visual Model* describes the structure of the target program. In its simplest form, it seems as an interactive code visualization that may assist in understanding the overall program source. It consists of a set of *Model Elements* described in the form of graphical notations that represent AO-basic programming constructs and features. Each element can be customized, and has a specification according to the syntactic formalism description of the corresponding concept. The whole specification of the model data is stored as a relatively small file (*.vaj), defined in the XMI format.

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

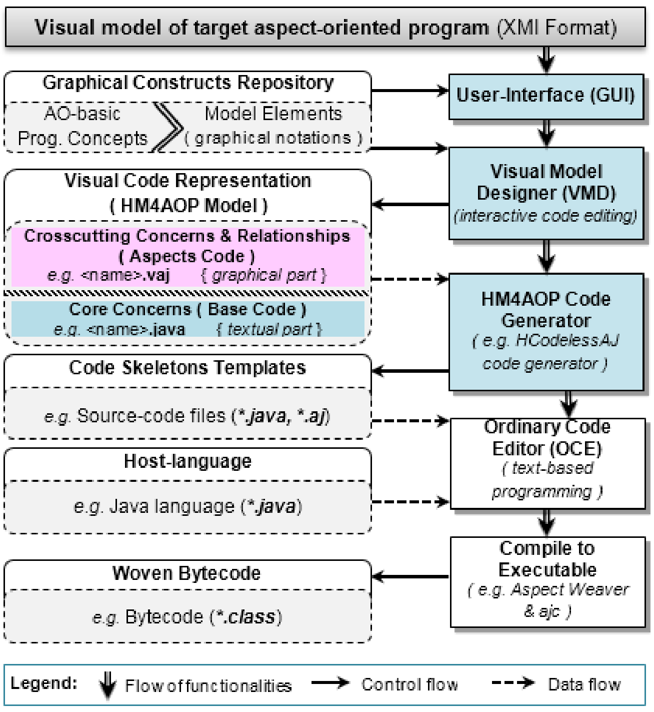

Visual model of target aspect-oriented program (XMI Format)
Graphical Constructs Repository
AO-basic Prog. Concepts
Model Elements (graphical notations)
User-Interface (GUI)
Visual Code Representation (HM4AOP Model)
Crosscutting Concerns & Relationships (Aspects Code)
e.g. <name>.vaj { graphical part }
Core Concerns (Base Code)
e.g. <name>.java { textual part }
Visual Model Designer (VMD) (interactive code editing)
Code Skeletons Templates
e.g. Source-code files (*.java, *.aj)
HM4AOP Code Generator (e.g. HCodelessAJ code generator)
Host-language
e.g. Java language (*.java)
Ordinary Code Editor (OCE) (text-based programming)
Woven Bytecode
e.g. Bytecode (*.class)
Compile to Executable (e.g. Aspect Weaver & ajc)
Legend:
Flow of functionalities
Control flow
Data flow

Fig. 5. Functional architecture of HM4AOP tool support
(e.g., HCodelessAJ prototype).

Figure 5 depicts an overview of the functional architecture of HM4AOP tool support, e.g., HCodelessAJ prototype. It is an arrangement of functions and their sub-functions and interfaces, which define the steps sequencing for both control flow and data flow throughout the coding process. The facing down arrows represent the flow or steps of functionalities. The horizontal arrow shows the control flow, while the horizontal dash arrow shows the data flow.

The visual paradigm capabilities can facilitate programming tasks by using explicit and intuitive representations to express various aspects and entities of source-code [9]. To this end, it is desirable to provide carefully designed graphical notations for fundamental AO-basic programming constructs and features, such as aspect-constructs (*pointcut*, *advice*, *inter-type declarations*, and *weave-time declarations*), class-constructs (*method*, *field*, etc.), inheritance, and structured constructs (*package*, *class*, *interface*, *aspect*, *loops*, *conditions*, etc.).

In more advanced implementation, it is good to let the program source take the form of a visual, interactive document [16]. In order to facilitate navigation, these representations must be coupled with efficient interaction techniques that permit to support two main functions: (1) controlling programming concepts through their corresponding constructs in a high-degree of flexibility; and (2) specifying parameters and properties for each model element selected. Based on this requirement, we have selected the following features to be implemented: (a) semi-separation among users and the language syntax; (b) no restrictions; and (c) an ordinary graphical user-interface. In this trend, the high-level descriptions encapsulate the underlying implementation technology adopted, which eases its replacement, e.g., replacing AspectJ implementation with AspectC++.

2) Implementation Overview

○ *Steps Followed to Implement the Approach*

The process followed to define the approach includes three main steps depicted in the right side of Figure 6 and listed below:

- **Defining a Meta-model for an AOP-based implementation.**

To this aim, we first chose a target AOP-based implementation, and then we defined a source meta-model (e.g., AspectJ.ecore, an EMF Ecore model for AspectJ) able to represent the design of visual model to be built for the intended program. An Ecore model (**.ecore*) is an EMF meta-model of the target language generated from its UML Profile (**.uml*). It is used to ensure that the output complies with the language (i.e., the language's syntax should be defined with this meta-model). In addition to the Ecore model already defined, a Generator model (**.genmodel*) associated and required to generate code (e.g., *AspectJ.genmodel*). As opposed to the Ecore model that holds only platform-independent information (i.e., *PIM*, *Platform-Independent Model*); the Generator model provides the platform-specific information (i.e., *PSM*, *Platform-Specific Model*).

- **Developing an editor for the defined Meta-model.**

A graphical editor was developed to enable programmers to build, view, and edit visual models, which are instances of the meta-model defined. The resulting tool is an advanced code editor (*VMD*) which supports drag-and-drop, cut-and-paste and so on.

- **Develop an engine for automatic code generation.**

As our proposal is a template-based approach, the suggested mechanism of mapping model-to-code is mainly based on the technique of Acceleo Model Transformation (*M2T*, *Model-to-Text transformation*) as a standard alternative for code generation [30]. This mapping is achieved by the application of M2T transformation rules expressed in *Acceleo modules* (**.mtl*) to automatically transform the visual model already constructed into *AO code skeletons templates* conforming to the chosen target AOP-based implementation. The code-generation engine component is composed of a collection of *Acceleo modules*, which are made of set of *Acceleo templates* (i.e., scripts to customize the generator accurately) that define the required rules of transformation. These templates are implemented by using Acceleo language within an EMF-based tool support. A valid template of the target code that determines its content must be defined and developed before establishing the process of code generation.

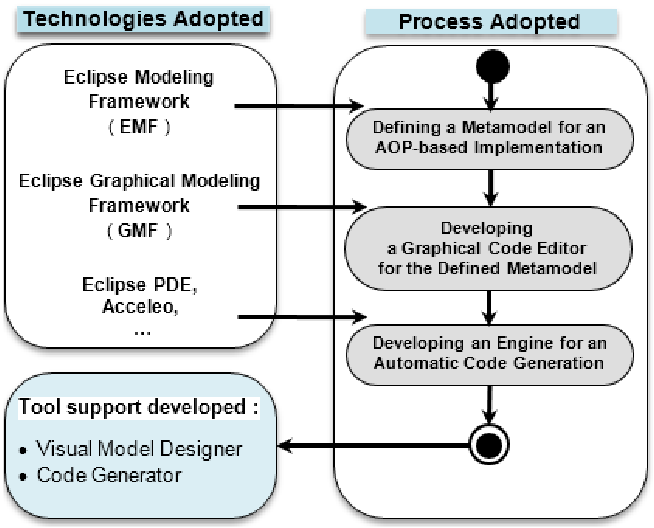

Technologies Adopted
Process Adopted
Eclipse Modeling Framework (EMF)
Eclipse Graphical Modeling Framework (GMF)
Eclipse PDE, Acceleo, ...
Defining a Metamodel for an AOP-based Implementation
Developing a Graphical Code Editor for the Defined Metamodel
Developing an Engine for an Automatic Code Generation
Tool support developed :
• Visual Model Designer
• Code Generator

Fig. 6. Overall process and technologies adopted for implementing HM4AOP tool support.

○ *MDE Technologies and Frameworks Adopted*

The underlying technology is the Eclipse platform with its plug-in capabilities. The approach exploits well known MDE (Model-Driven Engineering) technologies and frameworks provided under the Eclipse Modeling Project (EMP) [23,51], such as Eclipse EMF, GEF, GMF, and Acceleo technology [30], to enable building its graphical editor (*VMD*) besides a code-generation engine.

The left side of Figure 6 summarizes the frameworks and technologies we used. To define a metamodel for a target AOP-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* [30], for transforming models conforming to (i.e., an instance of) an EMF meta-model into text (source-code).

○ *The Transformation Process*

Figure 7 summarizes the process to generate code templates. In particular, it highlights the technologies adopted at each step. Our development process is based on the Model-Driven-Architecture (MDA). Part (A) shows the MDA process as described in [38]. Part (B) illustrates our development process as a translation service for generating templates.

The overall process is carried out in two main phases. First, it takes as input the visual model created, conforming to the EMF meta-model (e.g., *AspectJ.ecore*) and stored in XMI format file (*.vaj). Second, parsing and generating in the output templates of code skeletons (containing the structure and some behavior) on the target implementation, e.g., *AspectJ code templates*.

In the case of current prototype (Figure 5), the suitable *HM4AOP Code-Generator* is the *HCodelessAJ Code-Generator* component that uses the corresponding Acceleo modules and the Generator Model, e.g., *AspectJ.genmodel* to generate the code. The generated templates can be completed by adding the rest of required code to finalize the program. The resulting code is a complete AspectJ program that can be processed using the language compiler (ajc).

○ *HCodelessAJ, a first prototype tool*

Figure 8 shows a screenshot of the current stage of the tool on the Eclipse platform. The first goal is to show the feasibility of the approach. It works well for small programs but it has not yet been tested for the development of large real systems. A sample demonstration, showing an overview of the main features and how it essentially works, available online at: http://www.bentrad-sassi.sitew.com/.

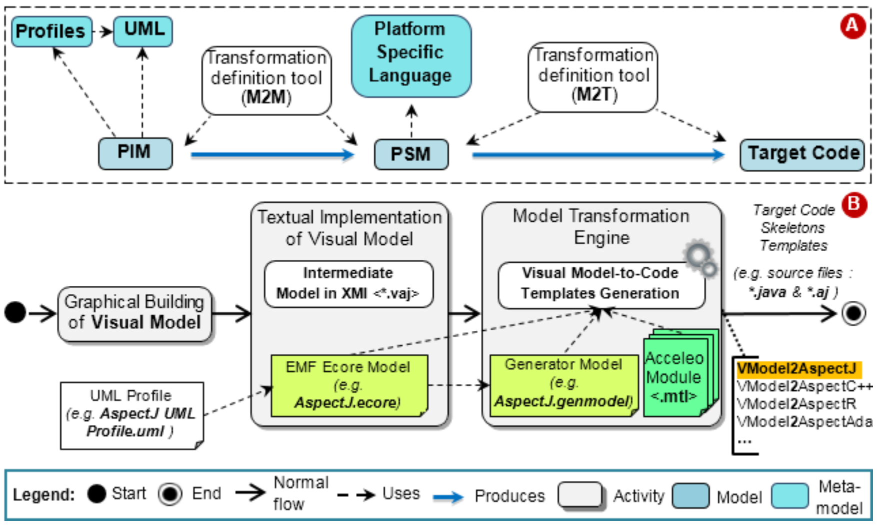
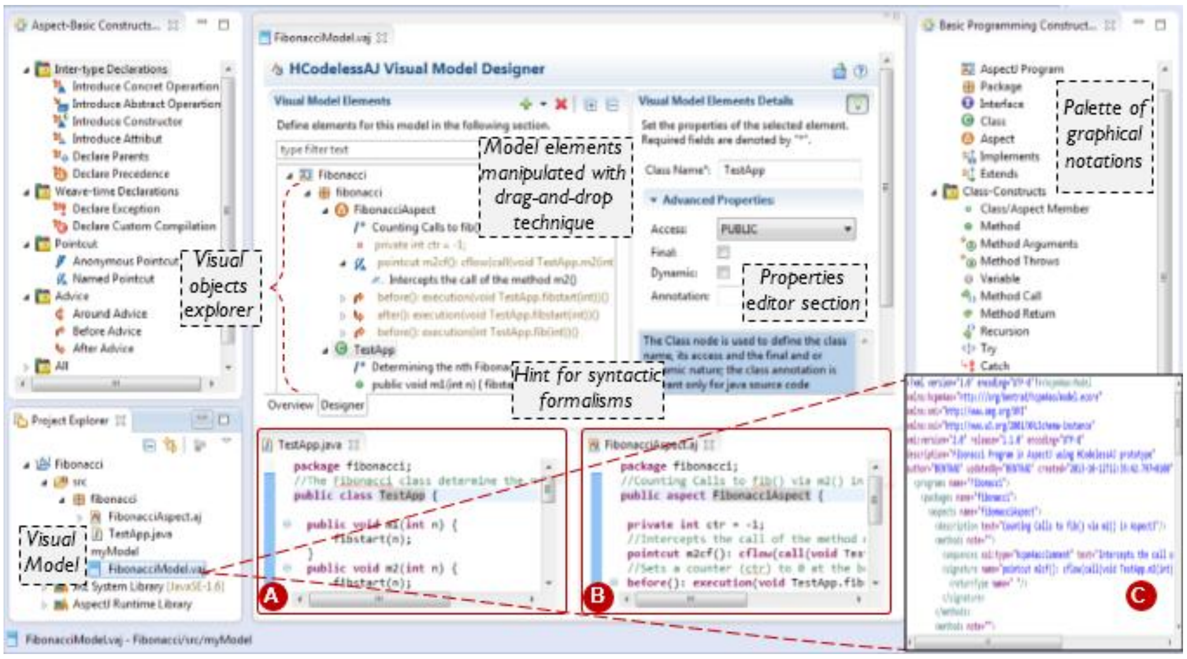

Profiles
UML
PIM
Transformation definition tool (M2M)
Platform Specific Language
PSM
Transformation definition tool (M2T)
Target Code
A
B
Graphical Building of Visual Model
Textual Implementation of Visual Model
Intermediate Model in XMI <*.vaj>
EMF Ecore Model (e.g. AspectJ.ecore)
UML Profile (e.g. AspectJ UML Profile.uml)
Model Transformation Engine
Visual Model-to-Code Templates Generation
Generator Model (e.g. AspectJ.genmodel)
Acceleo Module <.mtl>
Target Code Skeletons Templates
(e.g. source files : *.java & *.aj)
VModel2AspectJ
VModel2AspectC++
VModel2AspectR
VModel2AspectAda
...
Legend:
● Start
◉ End
Normal flow
Uses
Produces
Activity
Model
Meta-model

Fig. 7. Overall process of generating code templates.



```
package fibonacci;
//The fibonacci class determine the
public class TestApp {

  public void m1(int n) {
      fibstart(n);
  }
  public void m2(int n) {
      fibstart(n);
```

```
package fibonacci;
//Counting Calls to fib() via m2() in
public aspect FibonacciAspect {

 private int ctr = -1;
 //Intercepts the call of the method
 pointcut m2cf(): cflow(call(void Tes
 //Sets a counter (ctr) to 0 at the b
 before(): execution(void TestApp.fib
```

Fig. 8. HM4AOP tool support screenshot (e.g., HCodelessAJ prototype).
(A) & (B) snippets of fibonacci program implementation respectively in "VMD" and "OCE"; (C) a fragment of an XMI representation of visual model created.

IV. Evaluation

The first goal of undertaking an early assessment was to show the interactive capabilities of this prototype, as presented previously, and to investigate and evaluate the effectiveness of the approach.

A. Case Study

We report herein a preliminary concrete case study on an illustrative example of how the current version of the prototype can be used. We have considered a simple AspectJ project named "*Telecom example*", which is packed with the AJDT distribution [10,11,24]. It handles local and long distance phone connections between customers. An introductory video demonstration of this first example of use can be found on the author's website (see Appendix).

As a first step (*Rapid Code-Prototyping*), we complete the UML class diagram to indicate how the aspects intervene in the application. We indicate which aspects advice which methods (or calls methods) and, which aspects declare which members in which classes. Figure 9 shows the UML source-model.

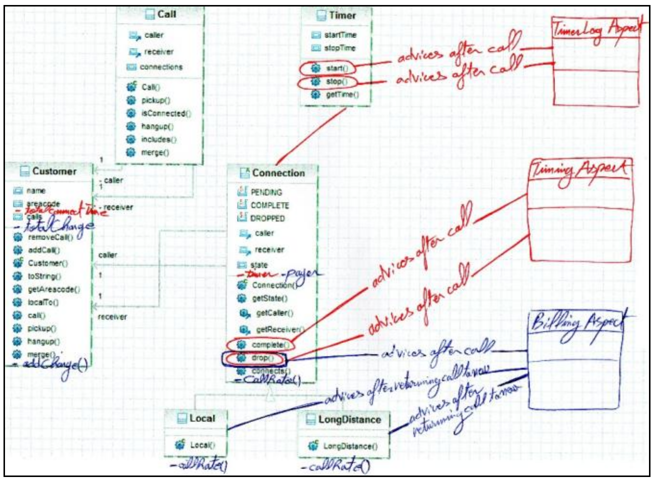


Fig. 9. Paper-based prototyping of aspects candidates.

The project has six classes: *Call*, *Timer*, *Customer*, *Connection*, *Local* and *Local Distance*.

– *Call*: A call supports the process of a customer trying to connect with others.

– *Timer*: A simple timer machine used to record elapsed time.

– *Customer*: Customers have a unique ID (name in this case for didactic purposes but it could be telephone number) and area code. They also have a protocol for managing calls.

– *Connection*: Connections are circuits between customers. There are two kinds: *Local* and *LongDistance* (subclasses).

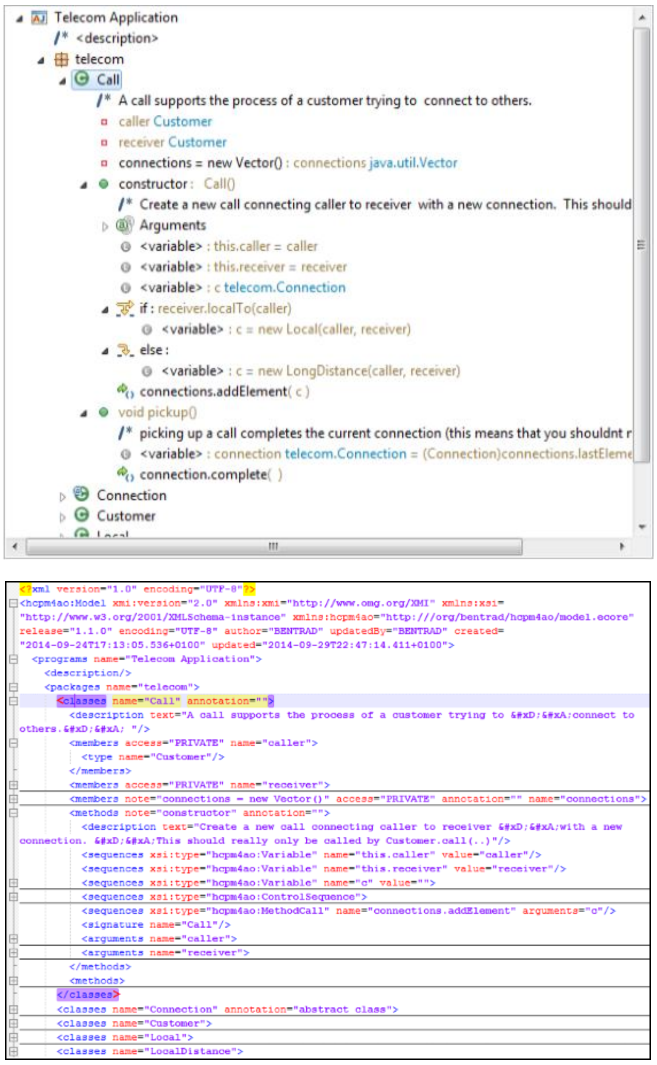
We decided to add three aspects:

– *Timerlong Aspect* is concerned with the connection time recorded.

– *Timing Aspect* is concerned with the duration of connections and with customer's cumulative connection time.

– *Billing Aspect*, to generate a bill according to the type of the call and the corresponding connection time.

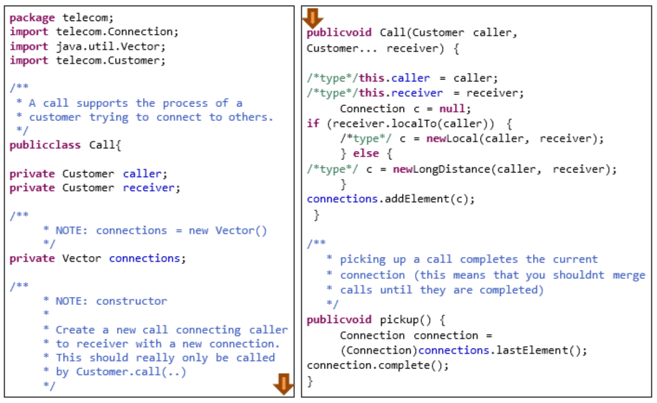
Then (*Interactive Code-Editing*), we express visually the code structure and some behavior for the target program using the visual designer. Figure 10 shows a snippet of the visual model created (*.vaj) and the corresponding XMI output definition file.



```
Telecom Application
  /* <description>
  telecom
    Call
      /* A call supports the process of a customer trying to  connect to others.
      caller Customer
      receiver Customer
      connections = new Vector() : connections java.util.Vector
      constructor :  Call()
        /* Create a new call connecting caller to receiver  with a new connection.  This should
        Arguments
        <variable> : this.caller = caller
        <variable> : this.receiver = receiver
        <variable> : c telecom.Connection
        if : receiver.localTo(caller)
          <variable> : c = new Local(caller, receiver)
        else :
          <variable> : c = new LongDistance(caller, receiver)
        connections.addElement( c )
      void pickup()
        /* picking up a call completes the current connection (this means that you shouldnt r
        <variable> : connection telecom.Connection = (Connection)connections.lastEleme
        connection.complete( )
    Connection
    Customer
    Local
```

```
<?xml version="1.0" encoding="UTF-8"?>
<hcpm4ao:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xmlns:hcpm4ao="http:///org/bentrad/hcpm4ao/model.ecore"
release="1.1.0" encoding="UTF-8" author="BENTRAD" updatedBy="BENTRAD" created=
"2014-09-24T17:13:05.536+0100" updated="2014-09-29T22:47:14.411+0100">
  <programs name="Telecom Application">
    <description/>
    <packages name="telecom">
      <classes name="Call" annotation="">
        <description text="A call supports the process of a customer trying to &#xD;&#xA;connect to
others.&#xD;&#xA; "/>
        <members access="PRIVATE" name="caller">
          <type name="Customer"/>
        </members>
        <members access="PRIVATE" name="receiver">
        <members note="connections = new Vector()" access="PRIVATE" annotation="" name="connections">
        <methods note="constructor" annotation="">
          <description text="Create a new call connecting caller to receiver &#xD;&#xA;with a new
connection. &#xD;&#xA;This should really only be called by Customer.call(..)"/>
          <sequences xsi:type="hcpm4ao:Variable" name="this.caller" value="caller"/>
          <sequences xsi:type="hcpm4ao:Variable" name="this.receiver" value="receiver"/>
          <sequences xsi:type="hcpm4ao:Variable" name="c" value="">
          <sequences xsi:type="hcpm4ao:ControlSequence">
          <sequences xsi:type="hcpm4ao:MethodCall" name="connections.addElement" arguments="c"/>
          <signature name="Call"/>
          <arguments name="caller">
          <arguments name="receiver">
        </methods>
        <methods>
      </classes>
      <classes name="Connection" annotation="abstract class">
      <classes name="Customer">
      <classes name="Local">
      <classes name="LocalDistance">
```

Fig. 10. An excerpt of a visual model and its output XMI definition for the target project.



```
package telecom;
import telecom.Connection;
import java.util.Vector;
import telecom.Customer;

/**
 * A call supports the process of a
 * customer trying to connect to others.
 */
publicclass Call{

private Customer caller;
private Customer receiver;

/**
     * NOTE: connections = new Vector()
     */
private Vector connections;

/**
     * NOTE: constructor
     *
     * Create a new call connecting caller
     * to receiver with a new connection.
     * This should really only be called
     * by Customer.call(..)
     */
```

```
publicvoid Call(Customer caller,
Customer... receiver) {

/*type*/this.caller = caller;
/*type*/this.receiver = receiver;
      Connection c = null;
if (receiver.localTo(caller)) {
      /*type*/ c = newLocal(caller, receiver);
      } else {
/*type*/ c = newLongDistance(caller, receiver);
      }
connections.addElement(c);
 }

/**
   * picking up a call completes the current
   * connection (this means that you shouldnt merge
   * calls until they are completed)
   */
publicvoid pickup() {
      Connection connection =
      (Connection)connections.lastElement();
connection.complete();
}
```

Fig. 11. Generated AspectJ code template.

Figure 11 illustrates the result of mapping the XMI description into code templates on the target implementation (e.g., AspectJ code). We just show the generated template of the indicated (the circled) model element presented in the Figure 10.

B. *User Evaluation*

We conducted a preliminary user evaluation to show the effectiveness of the proposed approach and to prove its feasibility. The experimentation was conducted on a group of sixteen students of Master degree in Software Engineering.

In order to perform an impartial analysis and to clarify the features of the prototype (HCodelessAJ) as compared to the ordinary AspectJ-based tool support for Eclipse IDE (AJDT), we asked them to build code for some examples of programs that are provided with the AJDT distribution using these tools separately after minimal training. The satisfaction of the users has been also investigated here. To this aim, the evaluation has been divided into three steps:

Firstly, the participants were given a short, formal introduction, providing them with the background and purposes, as well as a demonstration illustrating the main features of both tools. In the second step, participants were randomly divided into two balanced groups. We instructed both groups to write the source-code of some selected programs using separately both tools. One group started with AJDT, whereas the other had to build code firstly using HCodelessAJ. During every experiment moment, for each participant and each program, some observation data were annotated and written down with respect to some criteria (Pretest: *Controlled Experiment*). In the third step, the participants have to fill in a post-experiment questionnaire to collect information on their satisfaction, opinions and comments (Posttest: *Questionnaire-Based Survey*).

During every experiment moment, we did not provide any help to the participants to avoid biasing the experiment. We only wrote comments and difficulties they encountered. For each participant and each program, some observation data were annotated and written down. Finally, in order to perform an impartial analysis, we conducted a comparison of the obtained results (quantitative & qualitative), and the opinions gathered among participants.

In the following, we provide a summary of assessment tables (*Quantitative Assessment*).

TABLE 2
General Summary of the Quantitative Assessment Tables

Test Programs	**AJDT**				**HCodelessAJ**			
	T1	T2	T3	P	T1	T2	T3	P
Program 1								
. . .								
Program n								
Average								

Legend:
T1: Training time (in day)
T2: Development time (in minutes)
T3: Compilation time (in seconds, ms)
P: Miss-typing percentage (%)
NB: In this table, we show the average of results obtained from all participants.

The results of the comparative survey between AJDT and HCodelessAJ according to some defined criteria (*Qualitative Assessment*), are summarized in Table 3, and presented graphically in Figure 12.

TABLE 3
General Summary of a Comparative Study According to Qualitative Criteria

Aspects Considered / **AspectJ Tool Support**	**Expressiveness**	**Usability :** Efficiency, Learnability, Memorability, Error Handling, User Satisfaction	**Computational Performance**	**Productivity :** Flexibility, Scalability, Reusability, Comprehensibility, Maintainability
AJDT	2/4	2/4	4/4	2/4
HCodelessAJ	3,5/4	3/4	4/4	3,5/4

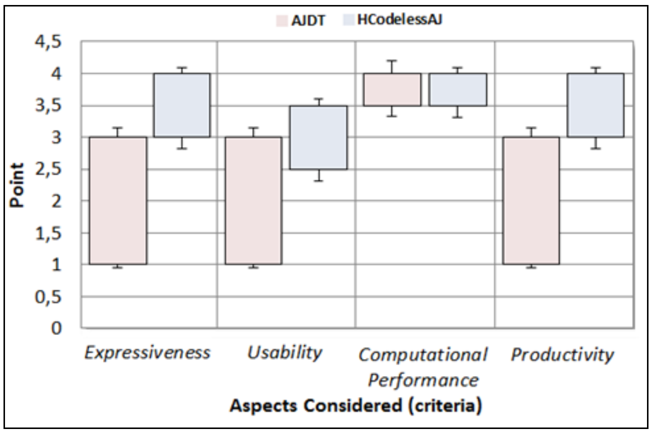


Fig. 12. Boxplots of the Questionnaire Answers.

The post-experiment questionnaire we used contained 26 questions, shown in Table 4, arranged in seven categories.

TABLE 4
The Posttest: A Questionnaire-Based Survey

ID	**Questions & Responses** (Percentage % : ?/16)
	Subject Experience (Background Information)
Q1	. . .
	Methodology of Coding & Prototype Satisfaction
Q11	. . .
	Performed Coding Task
Q12	. . .
	Suitability for the Coding Task
Q16	. . .
	Suitability for Learning
Q21	. . .
	Error Tolerance
Q26	. . .
	Comments . . .

C. *Discussion*

The results indicated the effectiveness of the tool support, even if in its prototype stage, for the development of small programs. As depicted in the visual editor, the representation of program code makes explicit some of its entities with their constructs and relationships (such as inheritance, crosscutting relationships, and more) which are implicit in the textual descriptions within conventional editors. The amount of code generation with respect to the total project code is about 70% of the considered examples. In addition, the high-quality templates are consistent with original program code, and easy to understand, which in turn improved the overall factors of quality such as extensibility, maintainability, and reusability.

Another point is that various capabilities of interactive and graphical techniques can make it easier to see a big picture of the system. Professional programmers and novices alike can get an idea of what the program does at a glance.

D. Threats to Validity

According to *Gregor Kiczales* and his team: "it is extremely difficult to quantify the benefits of using AOP without a large experimental study, involving multiple programmers using both AOP and traditional techniques to develop and maintain different applications" [17]. Since the case study was explorative in nature and aimed to illustrate and investigate the effectiveness of our proposal; the analysis is primarily qualitative, which can be considered as a threat to the validity. However, it appeared to be too small to observe a statistically significant difference compared to the conventional methodology.

Although the current prototype is still in its experimental stage, participants perceived that the proposed approach is usable and easy to use. The obtained results revealed a good satisfaction degree. Readers can return to the appendix when seeking information on the detailed evaluation.

From another point of view, think that adopting the approach will lead finally to an interesting alternative way of teaching an aspect-oriented language. Compared to AJDT, the prototype supports an alternative to the AJDT wizard for generating aspects, as it is more complete: can fill aspects with pointcuts, advices, introductions, and others. However, major challenges in this area include enhancing the code generation engine, developing a second engine for reverse engineering, and adding visual representations for various concepts and features borrowed from AOP.

The automated transformation through Acceleo is a growing area of interest due to its main benefits such assist possibility to generate behavioral specifications, the cost reduction and the accuracy of generated templates. Acceleo does not restrict the kind of code generated; there is only one rule: "*If you can write it, Acceleo can generate it*". In this way, the proposed approach could be adapted to work with various AOP-based implementations.

From our point of view, based on what has been presented herein, the maintainability will be higher because developers will have to update the visual models already build, generate new templates and transfer parts of an old source-code to new templates. As our contribution is the first step towards a novel support for aspect-oriented coding, there are still more possible extensions to do (we will discuss them in Section VI). After, some extensions, a large-scale study assessment is required in the context of real applications. In addition, it is thus important to evaluate how the approach affects the code quality and development efficiency.

V. Related Works

The graphical techniques have been around the software-development space for a long time and have also been investigated in a variety of fields. For the last decade, a great deal of research has been concerned with the development of new support tools that enable to accomplish tasks more quickly and effortlessly when compared to the traditional coding [5].

VP’s efforts have a long history and have been used successfully so far in many application areas from educational software to specialized programming tools [41,43]. However, only a few general-purpose tools are available currently. Some of them addressed the OOP paradigm such as *Prograph* [6], *Larch Environment* [7], *BlueJ* [29] and *Larch Environment* [7]. The latter is a based programming environment for Python that takes a hybrid approach; combine textual and visual programming by allowing visual, interactive objects to be embedded within textual source-code, and segments of code to be further embedded within those objects. Additionally, many of them are used exclusively for educational purposes (e.g., output syntactically correct code, etc.) as is the case of *Alice* [25], *Greenfoot* [28] and *Raptor* [34]. The *Limnor Studio* [26], *Tersus* [27] and *Jeeves* [33] are three mature tools that employ the drag-and-drop interaction style within a visual editor but for limited standalone, web and mobile applications.

AOP is an active research field, which has made huge progress both in theoretical and practical aspects [17]. Over the last few years, it has matured and received considerable attention from research and practitioner communities alike, where numerous works have been carried out on AOP-based implementations such as the case of AspectJ, which is the major language available to practitioners [2,10]. To the best of our knowledge, although a significant number of works focused on AOP as a promising field, among a wide variety of support tools, there has been no work performed upon various capabilities of interactive and graphical techniques.

Among the projects that were sources of inspiration for our work were *Citrus* [31], *Barista* [32] and *Limnor Studio* [26]. The latter is one of the most experimental tools based on the idea of general-purpose codeless program development, which can be used to build efficiently programs in an interactive manner. We have emulated its interaction technique, but in a new way. We have focused to merge graphical techniques together with the conventional style as a practical option of coding. In this respect, we can reduce the hand-typing at development time.

Our choice of AspectJ was influenced by its maturity and availability of good documentation. The implemented prototype, in the form of Java plug-ins, completely leverages the AJDT project [10,24]. We have already positioned and motivated our work with respect to AJDT; as it is arguably the most complete, open mature tool support and regarded as the representative in the community for this language [32].

Our tool essentially distinguishes itself from it in that, within the visual editor, the new form of representation of code makes explicit some of its entities, its code structure and their relationships, which are implicit in the textual descriptions within conventional editors as is the case with AJDT. Hence, program source may be much easier to read and understand and, therefore, to maintain, to extend it and facilitating its reuse.

From a state of the art review in model transformations and code generation, many proposals have been identified. The most relevant for AOP is introduced in [48], by *J. Bennett, K. Cooper* and *L. Dai*. They proposed a code generation approach from models to aspects. The translation among models is accomplished by means of XML specifications and metamodels of XML and AspectJ. The code is generated from those specifications and the aspects are controlled in the program by throwing and handling exceptions.

In [12,13], *Abid Mehmood* and *Dayang N.A. Jawawi* conducted a systematic mapping study of existing research in the area of aspect-oriented model-driven code generation. The existing methods focus on UML design models with just the possibility to generate code skeletons. The most related work to our adopted mechanism is the work done by *Hyun Seung et al*. [49]. They proposed an approach to generate more sophisticated Java code for Android Platform from both *UML Class* and *Message Sequence diagrams*, based on Acceleo. Until now, our mapping mechanism deals with an automatic generation of Java/AspectJ code templates from the visual model data, already build and stored in an XMI format, by using predefined Acceleo transformation rules.

VI. Conclusion And Future Work

A. Conclusion

Despite the evolution of programming paradigms, the traditional style is still disheartening and boring. Users suffer generally from some technical limitations due to the input method adopted for code editing which reduces the productivity and software quality. In fact, for the case of AOP, a great challenge for editing arises, where usually most of the abstractions that are meaningful at the design phase may be lost at implementation time due to the lack of solid and appropriate support tools. With new language constructs that are not straightforward for novices, AOP offers new ways to implement traditional mechanisms. In our opinion, certainly this area needs further research.

To overcome that and in order to make AOP more widely accepted, we proposed an approach for expressing and specifying crosscutting concerns. It is partially visual that leverages the usage of interactivity for providing a degree of freedom with respect to source-code languages and their syntactic formalisms while overcoming some weaknesses in the traditional style.

To show the feasibility of our approach, a prototype tool was developed and a preliminary evaluation conducted to assess its usefulness. It seems highly promising to narrow the gap between the programmer and the programming system, and enable an effective coding by focusing more on innovations instead of on implementation details.

In general, we believe that the programming environments will change greatly in the future according to our proposal. This is expected to improve programming motivation, program comprehension, and efficiency of programming education. In other way, it is also promising for teaching techniques, especially in the area of AOP. We also believe it is essential for supporting novices to take the paradigm into their own hands.

AOP is promising and deserves more attention from developers and researchers. This work can be considered as a step towards making this paradigm more usable and more efficient for development. Currently, the tendency has been emphasized to introduce it into programming courses for undergraduate students. We hope that it succeeds to attract a considerable interest from researchers and practitioners. In addition, we hope that the planned extensions, mentioned below, are appropriate for emerging further research.

B. Future Work Issues

Other ongoing work is intended to improve the approach itself. We could divide possible extensions from the current state of research in three parts:

- The first opportunity lies in extending the current prototype with more graphical views including a set of interaction possibilities through advanced techniques that allow a user to easily edit and navigate the code (depicted in item (A) in Figure 13).
- Second, supporting multiple AOP-based implementations similar to AspectJ; like AspectC++, AspectR, AspectAda, etc. Such an extension to the editor "*VMD*" and their notations along with the "*HM4AOP Code Generator*" should be straightforward; only the mapping rules for code generation should have to be specified (we can use multiple metamodels in an Acceleo module, each one for an implementation). The generation templates on the preferred implementation from the generic visual model is performed by the suitable component within this code-generation engine (depicted in item (B) in Figure 13).

  This potential extension can contribute to:
  - Increasing the reusability of an AOP code through exporting its visual model for future reuse.
  - Enable easy interchanging of visual model data between various implementations—towards a neutral exchange based on XMI format among aspect-oriented software development tools.
- Third, while the translation service mechanism is mainly based on the Acceleo technology (*M2T, Model-to-Text transformation*) for transforming model elements into textual templates, it will be interesting to focus on doing the opposite (*Text-to-Model transformation*) with a reverse-engineering engine for the reconstruction of initial visual models from code (i.e., back transformation of AspectJ code).

The above extensions appear to be feasible and, in our opinion, will be useful for AOP. The programmer will always have the choice to switch seamlessly between both Bottom-Up and Top-Down paths whenever needed to transform the source of a program (low-level representation) from one implementation to another by creating visual models (higher-level models and artifacts) along the way and without performing major modification in the code. Ultimately, our goal is an attempt to introduce a complete tool supporting “Round-Trip Engineering" of produced artifacts, so that manual customizations of the generated templates are taken into account and merged by the generator (shown as item (C) in Figure 13).

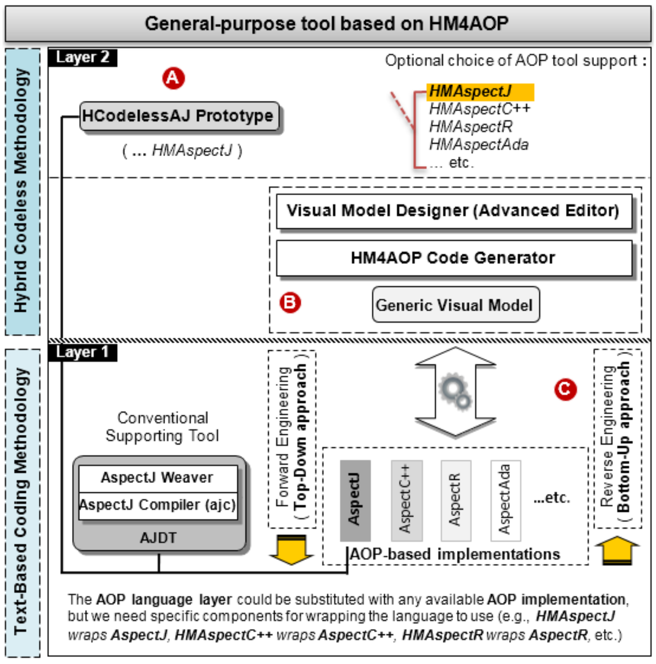

General-purpose tool based on HM4AOP
Hybrid Codeless Methodology
Layer 2
A
HCodelessAJ Prototype
(... HMAspectJ)
Optional choice of AOP tool support :
HMAspectJ
HMAspectC++
HMAspectR
HMAspectAda
... etc.
Visual Model Designer (Advanced Editor)
HM4AOP Code Generator
B
Generic Visual Model
Text-Based Coding Methodology
Layer 1
Conventional Supporting Tool
AspectJ Weaver
AspectJ Compiler (ajc)
AJDT
Forward Engineering (Top-Down approach)
C
AspectJ
AspectC++
AspectR
AspectAda
...etc.
AOP-based implementations
Reverse Engineering (Bottom-Up approach)
The AOP language layer could be substituted with any available AOP implementation, but we need specific components for wrapping the language to use (e.g., HMAspectJ wraps AspectJ, HMAspectC++ wraps AspectC++, HMAspectR wraps AspectR, etc.)

Fig. 13. A future vision of the HM4AOP framework architecture.

Figure 13 depicts a general preview of the conceptual architecture of a HM4AOP framework that collects these prospect extensions. As shown, in order to get flexibility for further extensions, we have divided its design into two main layers, which also could easily be enhanced with additional VP capabilities.

Appendix

Supplementary data, availability information, additional material: an online appendix, available on the author’s website (http://www.bentrad-sassi.sitew.com/), showing:

(1) A video demonstrations showing the main features and basic functionalities of the prototype "HCodelessAJ", along with brief examples of use.

(2) A detailed report on the results of a preliminary user evaluation we conducted.

Acknowledgment

The authors are highly grateful to the Referees and Editor-in-Chief for their valuable comments and suggestions helpful in improving this paper.

References

[1] Mens T. and Tourwé T., "Evolution issues in aspect-oriented programming," in: *Software Evolution*. Springer, Berlin, Heidelberg, pp203-232, 2008.
[2] Ramnivas Laddad, "AspectJ in action: practical aspect-oriented programming," *Manning Publications Co*., 2003.
[3] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W.G., "An overview of AspectJ," *Lecture Notes in Computer Science: Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, vol. 2072. Springer, Berlin, Heidelberg, pp327-353, 2001.
[4] Plauska, I. and Damasevicius, R., "Usability analysis of visual programming languages using computational metrics," in: *Proceedings of the IADIS International Conference on Interfaces and Human-Computer Interaction*, Chech Republic, pp63-70, 2013.
[5] Sanders K. and Van Dam A., "Object-oriented programming in Java: a graphical approach," Addison-Wesley, 2006.
[6] R. Mark Meyer and Tim Masterson, "Towards a better visual programming language: critiquing Prograph's control structures," *Journal of Computing Sciences in Colleges*, vol.15, no .5, pp181-193, 2000.
[7] French, G.W., "The Larch Environment - Python programs as visual, interactive literature," *Master of Science Thesis*, School of Computing Science - University of East Anglia, 2013.
[8] Ferruci F., Tortora G., and Vitello G., "Exploiting visual languages in software engineering," in: Chang S. K., *Handbook of Software Engineering and Knowledge Engineering*. River Edge, NJ: Singapore World Scientific, 2002.
[9] Zhang K., "Visual languages and applications," Springer-Verlag US, 2007.
[10] A. Colyer and A. Clement, "Aspect-oriented programming with AspectJ," in: *IBM Systems Journal*, vol. 44, no. 2, pp301-308, 2005.
[11] Colyer A., Clement A., Harley G., and Webster M., "Eclipse AspectJ: aspect-oriented programming with AspectJ and the Eclipse AspectJ Development Tools," Addison-Wesley Professional, 2004.
[12] Abid M. and Dayang N. A. Jawawi, "Aspect-oriented code generation for integration of aspect-orientation and model-driven engineering," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 2, 2013.
[13] Abid M. and Dayang N. A. Jawawi, "Aspect-oriented model-driven code generation: a systematic mapping study," *Information and Software Technology*, vol. 55, no. 2, pp395-411, 2013.
[14] Garcia Perez-Schofield J.B., Garcia Rosello E., Ortin Soler F., Perez Cota M., "Visual Zero: a persistent and interactive object-oriented programming environment," *Journal of Visual Languages & Computing*, vol. 19, no. 3, pp380-398, 2008.
[15] Despi I. and Luca L., "Aspect-oriented programming challenges," *Anale Seria Informatica*, vol. 2, no. 1, pp65-70, 2005.
[16] French, G.W., Kennaway, J.R., and Day, A.M., "Programs as visual, interactive documents," *Software: Practice and Experience*, vol. 44, no. 8, pp911-930, 2014.
[17] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J-M., Irwin, J., "Aspect-oriented programming," published in: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, Finland. Springer-Verlag LNCS 1241, pp220-242, 1997.
[18] L. Madeyski and L. Szala, "Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study," *IET Software Journal*, vol. 1, no. 5, pp180-187, 2007.
[19] Md. Asraful Haque, "Problems in aspect-oriented design: facts and thoughts," *International Journal of Computer Science Issues*, vol. 8, no. 2, 2011.
[20] D. Zhengyan, "Aspect-oriented programming technology and the strategy of its implementation," in: *Proceedings of International Conference on Intelligence Science and Information Engineering*, pp457-460, 2011.
[21] Roger T. Alexander and James M. Bieman, "Challenges of aspect-oriented technology," in: *ICSE Workshop on Software Quality*, Florida, 2002.
[22] Aspect-Oriented Software Development (AOSD), http://aosd.net/
[23] Eclipse Modeling Project (EMP). http://www.eclipse.org/modeling/
[24] Eclipse AJDT Project, https://www.eclipse.org/ajdt/
[25] Alice Project, http://www.alice.org/
[26] Limnor Studio, http://www.limnor.com/
[27] Tersus Project, http://www.tersus.com/
[28] Greenfoot Project, https://www.greenfoot.org/
[29] BlueJ Project, https://www.bluej.org/

[30] Acceleo Project, http://www.eclipse.org/acceleo
[31] Ko, A.J. and Myers, B.A., "Citrus: a toolkit for simplifying the creation of structured editors for code and data," in: *ACM Symposium on User Interface Software and Technology (UIST)*, pp3-12, 2005.
[32] Ko, A. J. and Myers, B. A., "Barista: an implementation framework for enabling new tools, interaction techniques and views in code editors," in: *Proceedings of Conference on Human Factors in Computing Systems*, vol. 1, pp387-396, 2006.
[33] Rough, D. and Quigley, A., "Jeeves – a visual programming environment for mobile experience sampling," in: *Visual Languages and Human-Centric Computing (VL/HCC)*, 2015.
[34] M. C. Carlisle, "Raptor: a visual programming environment for teaching object-oriented programming," *Journal of Computing Sciences in Colleges*, vol. 24, no. 4, pp275-281, 2009.
[35] Nong Ye and Gavriel Salvendy, "Expert-novice knowledge of computer programming at different levels of abstraction," *Ergonomics*, vol. 39, no. 3, pp461-481, 2007.
[36] F. Steimann, "The paradoxical success of aspect-oriented programming," in: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM, pp481-497, 2006.
[37] Muhammad Sarmad Ali, Muhammad Ali Babar, Lianping Chen, Klaas-Jan Stol, "A systematic review of comparative evidence of aspect-oriented programming," *Information and Software Technology*, vol.52, no.9, pp871-887, 2010.
[38] A. Kleppe, J. Warmer, and W. Bast., "MDA explained, the model-driven architecture: practice and promise," Addison-Wesley, 2003.
[39] Khin Zaw, Win Zaw, Nobuo Funabiki, and Wen-Chung Kao, "An informative test code approach in code writing problem for three object-oriented programming concepts in java programming learning assistant system," *IAENG International Journal of Computer Science*, vol. 46, no. 3, pp445-453, 2019.
[40] Rémi Dehouck, "The maturity of visual programming," 2015. [Online]. Available at : http://www.craft.ai/blog/the-maturity-of-visual-programming/
[41] F. I. Anfurrutia, A. Álvarez, M. Larrañaga and J. López-Gil, "Visual programming environments for object-oriented programming: acceptance and effects on student motivation," *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, vol. 12, no. 3, pp124-131, 2017.
[42] Aleksandr Miroliubov, "Visual programming – an alternative way of developing software," *Thesis Bachelor of Engineering: Information and Communications Technology*, Metropolia University of Applied Sciences, 2018.
[43] Gábor Csapó, "Placing event-action-based visual programming in the process of computer science education," *Acta Polytechnica Hungarican*, vol. 16, no. 2, University of Debrecen, 2019.
[44] Jeremy T. Bradley, "An examination of aspect-oriented programming in industry," *Technical Report*, Colorado State University, USA, 2003.
[45] G. Lommerse, F. Nossin, L. Voinea, and A. Telea, "The visual code navigator: an interactive toolset for source code investigation," in: *IEEE Symposium on Information Visualization (INFOVIS)*, Minneapolis, MN, pp24-31, 2005.
[46] Zhang, K., Kong, J., and Cao, J., "Visual software engineering," *Wiley Encyclopedia of Computer Science and Engineering*, 2007.
[47] P. Ahmed and S. Ahmadi, "Extended visual object based intelligent visual programming environment," in: *Proceedings of the 4th International Conference on Information and Automation for Sustainability (ICIAFS)*, Colombo, pp224-229, 2008.
[48] J. Bennett, K. Cooper and L. Dai, "Aspect-oriented model-driven skeleton code generation: a graph-based transformation approach," *Science of Computer Programming*, Elseiver, vol. 75, no. 8, pp689-725, 2010.
[49] H.S. Son, W.Y. Kim, and R.Y.C. Kim, "MOF based code generation method for Android platform," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 3, 2013.
[50] G.C. Murphy, R.J. Walker, and E. L. A. Banlassad, "Evaluating emerging software development technologies: lessons learned from assessing aspect-oriented programming," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp438-455, 1999.
[51] J. Bezivin, "Model driven engineering: an emerging technical space," *Lecture Notes in Computer Science: Generative and Transformational Techniques in Software Engineering (GTTSE)*, vol. 4143, pp36-64. Springer Berlin, Heidelberg, 2006.
[52] Cristina Videira Lopes, "Aspect-oriented programming: an historical perspective (what's in a name?)," *ISR Technical Report, Institute for Software Research*, University of California, Irvine, 2002.

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. His research interests include Software Visualization, Visual Programming, Model-Driven Engineering, and Separation of Concerns.

Hasan KAHTAN KHALAF is Lecturer with the Faculty of Computer Systems & Software Engineering, Universiti Malaysia Pahang (UMP), Malaysia. Currently, he is a member at the Software Engineering Research Group (SERG). His research interests include Software Security and Dependability Attributes, Fuzzy Logic Approach, Visualization, and Big Data.

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.